Escuela Superior de
Ingenieros de
San Sebastián

TECNUN

APRENDA A PROGRAMAR COMO SI ESTUVIERA EN PRIMERO

Iker Aguinaga
Gonzalo Martínez
Javier Díaz

ÍNDICE

INTRODUCCIÓN AL MANUAL

Este manual está dirigido en primer lugar a los alumnos que cursan primero en la escuela de Ingenieros de San Sebastián y se encuentran realizando la asignatura de ***Informática 1*** o ***Fundamentos de Computadores***.

Pretende servir para que las personas nóveles se introduzcan en la lógica de la programación y, someramente, en los lenguajes de programación.

Es un libro esencialmente práctico. No porque los ejemplos que en él aparecen sirvan para algo, sino porque rehuye de posibles elucidaciones abstractas -aun siendo estas tan necesarias para el desarrollo de la lógica computacional.

Es un manual que complementa a la colección de manuales ***"Aprenda Informática como si estuviera en Primero"*** publicados en TECNUN. Estos manuales son muy útiles para introducirse en el lenguaje de programación del que versan. Sin embargo, este manual no está orientado a aprender ningún lenguaje de programación, sino a incentivar al lector para que aprenda a razonar los pasos necesarios para realizar una tarea acertadamente. Esto es, para que aprenda a programar.

El sustrato de este manual es la experiencia de los autores y los libros clásicos de algoritmia. Los ejemplos que se presentan plantean problemas que no necesitan un conocimiento profundo de matemáticas para entender su planteamiento y resolución. Son, en muchos casos, ejemplos universalmente conocidos.

La primera parte del manual, hasta el capítulo 4 incluido, esboza el estado del arte y expone los rudimentos necesarios para la programación. Se enseñan los elementos comúnmente utilizados en distintos lenguajes; principalmente, cómo se estructura la información y como se controla el flujo de un programa. El capítulo 5 presenta una agrupación de algoritmos con ejemplos. Como se ha comentado anteriormente, no es intención del manual tratar con lenguajes de programación, pero la necesidad de aplicar los algoritmos que se presentan hace inevitable recurrir a éstos. En concreto nos hemos decantado por utilizar Matlab debido a sus simplificadoras ventajas.

Tipografía utilizada

Para indicar...	**Este manual utiliza...**	**Ejemplo**
Variables	**Negrita**	**M** es una matriz
Líneas del programa	**Negrita con tamaño de letra pequeño**	**i=i+1**
Funciones	***Negrita e Itálica***	La función ***sin(Fi)*** devuelve el seno del ángulo **Fi.**

Capítulo 1 Los programas

1.1 ¿Qué es un programa?

Un programa de ordenador es una secuencia de instrucciones que el ordenador debe seguir para realizar una tarea. Habitualmente, aunque no obligatoriamente, estas instrucciones se aplican sobre un conjunto de datos que sirven como entrada para el programa, y produce como resultado otra serie de datos que se derivan de los primeros al aplicar sobre los datos de entrada las instrucciones.

Debido a las capacidades de los actuales microprocesadores digitales, absolutamente todos los datos se representan de forma numérica y digital. Aunque esto pueda parecer inicialmente una limitación es en realidad la causa de la enorme flexibilidad y poder de las modernas computadoras.

1.2 ¿Qué es un lenguaje de programación?

Como los lenguajes humanos, los lenguajes de programación son herramientas de comunicación, pero al contrario que los lenguajes corrientes como el inglés o el chino, los destinatarios de los lenguajes de programación no son sólo humanos sino también los ordenadores.

El propósito general de un lenguaje de programación es permitir a un ser humano (el programador) traducir la idea de un programa en una secuencia de instrucciones que el ordenador sea capaz de ejecutar.

1.3 Las cuatro patas de la programación

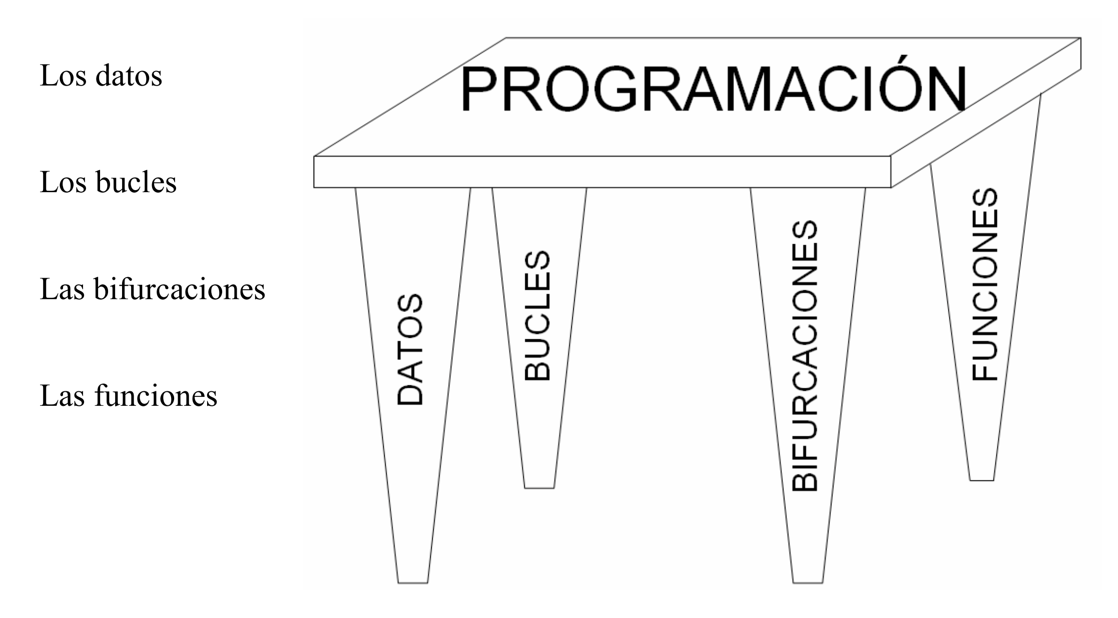
Las cuatro patas de la programación son:

Los datos

Los bucles

Las bifurcaciones

Las funciones



Los datos que almacena un ordenador son siempre números. Incluso las letras almacenadas en un ordenador se almacenan como números; en este caso los números codifican una letra (por ejemplo, la letra "a" es el número 97 en la codificación ASCII). Los datos se pueden almacenar ordenadamente como:

Datos individuales

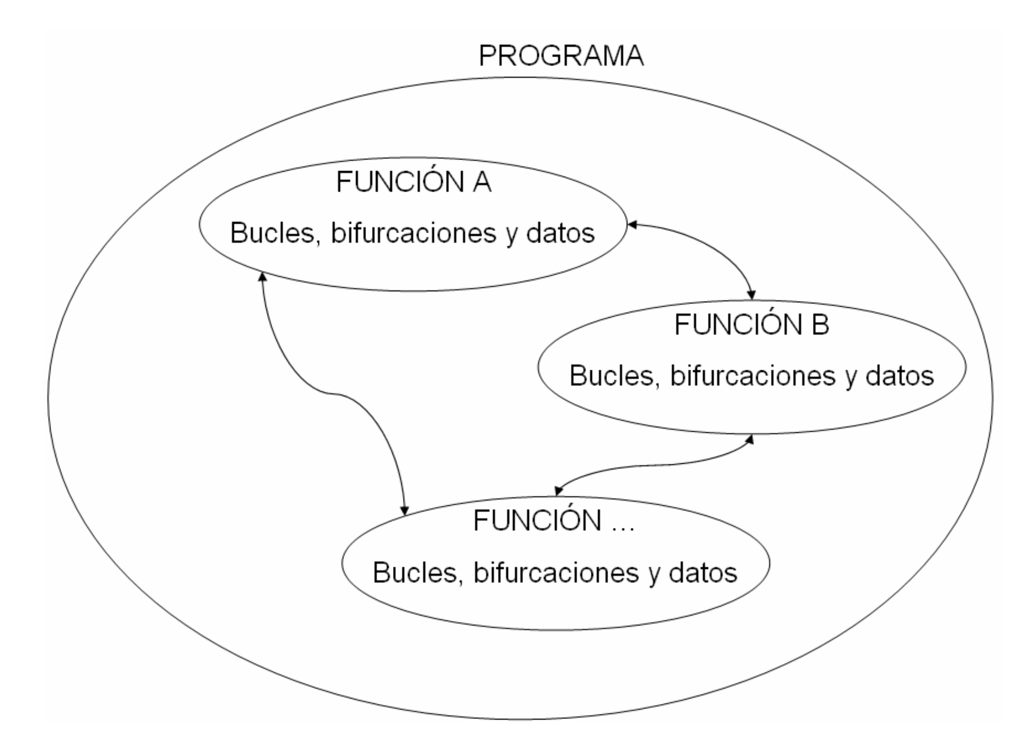
Vectores

Matrices

Hipermatrices

Jugando con estos números se consiguen hacer los programas. Las herramientas de las que disponen los lenguajes de programación para jugar con los datos son los bucles y las bifurcaciones, con quienes se controla el flujo del programa.

Por último hay que destacar que la programación exige orden. El orden se consigue separando las distintas subrutinas que componen el programa y esto se consigue mediante las funciones. El conjunto de las distintas funciones y subrutinas conforman el programa.



1.4 Lenguajes de programación

Existe una cantidad gigantesca de lenguajes de programación distintos (incluidos muchos dialectos), y muchas formas de clasificarlos. A continuación se verán algunas de las más importantes y se nombrarán algunos de los lenguajes más populares, o importantes, en cada una de las categorías.

1.4.1 Lenguajes de alto y bajo nivel

Esta clasificación divide a los lenguajes según la proximidad de las instrucciones que emplea el programador a las instrucciones que físicamente emplea el procesador de una computadora. Estas últimas son en general muy sencillas y tienen un valor numérico que las define. Aunque es posible crear un programa empleando directamente estos valores numéricos, en cuanto un programa alcanza unas pocas decenas de instrucciones comienza a ser completamente inmanejable.

Para facilitar el trabajo de los programadores todos los fabricantes de CPU crean lenguajes específicos para las instrucciones que soportan sus microprocesadores, sustituyendo los valores numéricos de las instrucciones por un pequeño identificador o mnemónico. Por ejemplo, los procesadores de Intel o AMD que encontramos en los PC comparten las instrucciones y el lenguaje de ensamblador, pero no así el procesador de una SONY Playstation que es distinto a los otros dos mencionados y requiere de otro lenguaje de ensamblador.

Debido a la cercanía con las instrucciones básicas, los lenguajes de ensamblador se consideran lenguajes de bajo nivel. Los lenguajes de alto nivel por el contrario abstraen al programador de las instrucciones, mediante el uso de expresiones más cercanas al lenguaje natural y/o a las matemáticas. Como botón de muestra compare estos ejemplos escritos en lenguaje ensamblador y con Matlab para realizar la misma operación:

Ensamblador para procesadores Pentium o AMD Athlon	Matlab
mov eax, 10; mov ebx, 12; add eax, ebx; mov r, eax;	**r=10+12**

1.4.2 Lenguajes imperativos y funcionales

En los lenguajes de programación imperativos los programas se dividen claramente en datos y código, siendo este último el encargado de manipular los datos de forma explícita. Suelen basarse, por lo tanto, en la manipulación de variables a las que el programa les va asignado valores específicos.

En los lenguajes funcionales, por el contrario, se diluye en cierta medida la diferencia entre código y datos al poder asignar bloques de código a variables.

Compárese una función para calcular el factorial en el lenguaje Scheme (funcional) y en Matlab (imperativo), se puede comprobar que en el caso de Matlab existen variables a las que se le asignan valores directamente (mediante el operador de asignación =) y la inexistencia del mismo en Scheme.

Scheme	Matlab

(define fact (lambda (n) (if (> n 1) (* n (fact (- n 1))) 1)))	**function f=fact(n)
	f=1;
	for i=1:n
	   f=f*i;
	end**

Algunos lenguajes funcionales son Lisp, Scheme, OCaml, Haskell, etc.

Entre los lenguajes imperativos encontramos Matlab, C, C++, Pascal, Java, Basic, Fortran, etc.

1.4.3 Lenguajes interpretados y compilados

El código de un programa suele escribirse en uno o varios ficheros de texto. Este código que es interpretable por un humano (o al menos debiera serlo) no puede ser ejecutado directamente por el computador.

En los lenguajes interpretados, existe un programa llamado intérprete que lee las líneas de código y las ejecuta inmediatamente. Muchos intérpretes admiten dos modos de empleo: interactivo y de script. En el primer caso, el intérprete muestra un símbolo y se queda esperando a que el usuario introduzca las líneas a ejecutar una a una, ejecutándolas inmediatamente. En el segundo, el usuario proporciona uno o más ficheros al intérprete que ejecutará todas las líneas una tras otra. Matlab es un lenguaje interpretado que admite estas dos formas de ejecución, la primera se realiza escribiendo el código a ejecutar en la ventana de comandos, mientras que la segunda forma es la que se lleva a cabo al emplear ficheros .m. En cambio Visual Basic, que también es un lenguaje interpretado, sólo admite la segunda forma.

Aunque los lenguajes interpretados suelen ser relativamente sencillos de emplear y son muy flexibles, no suelen ser muy eficientes, ya que se mezcla la tarea de interpretación y proceso del código con la ejecución de este código. Para evitar este problema y que los programas puedan aprovechar completamente los recursos de un ordenador, el código de un programa puede ser convertido de forma completa en código binario que el procesador pueda ejecutar directamente. Este proceso se denomina compilación y se emplea en lenguajes que requieran un gran rendimiento como C++.

Además se han desarrollado algunos lenguajes, en particular Java y C#, que en lugar de compilar el código a la forma binaria que emplea el procesador concreto de una determinada arquitectura, se compila en un código intermedio no ejecutable directamente. Posteriormente, este código intermedio se traduce a instrucciones que el procesador puede ejecutar directamente antes de ser ejecutado. Este sistema permite ejecutar programas en cualquier procesador aunque tenga distintas instrucciones[1] siempre que exista el programa que sea capaz de traducir las instrucciones intermedias. En cualquier caso esta flexibilidad tiene un precio, ya que el rendimiento de un programa escrito en uno de estos lenguajes siempre es menor que el de un programa compilado. No obstante, lenguajes con la versatilidad de interpretación como la que presenta Java, resultan particularmente interesantes para aplicaciones en Internet, donde van a interactuar distintos procesadores a través de la red.

1.4.4 Lenguajes que soportan la programación estructurada

Estos lenguajes son lenguajes de alto nivel que ofrecen una serie de construcciones de control del flujo del programa como son:

- Las bifurcaciones, donde un bloque de código se ejecutará dependiendo de una condición.

[1] Las instrucciones de los x86 de los AMD Athlon o Intel Pentium, son distintas e incompatibles con las de los procesadores PowerPC que se encuentras en servidores de IBM, los Macintosh de Apple o las futuras Playstation 3 o X Box 2, y a su vez estas son incompatibles con los procesadores Sparc de las estaciones de trabajo de Sun Microsystems, los MIPS de las estaciones de trabajo de Silicon Grafics, los Intel Xscale de los PDA Pocket PC, o de los procesadores Intel Itanium que también se emplean en estaciones de trabajo y servidores.

- Los bucles, donde un bloque de código se ejecutará de forma repetida o bien mientras que se cumpla una condición (WHILE) o bien un número determinado de veces (FOR).
- Las funciones, que son bloques autónomos e independientes de código que se encargan de realizar una operación concreta, y que pueden recibir datos de otras partes del programa en forma de parámetros y que pueden devolver uno o más valores al terminar de ejecutarse.

Entre los lenguajes que soportan la programación estructurada se encuentra Matlab, C, C++, Fortran, Pascal, Modula, etc.

1.4.5 Lenguajes fuertemente y débilmente "tipados"

Bajo este título tan extraño se encuentra una de las diferencias más deteminantes entre los distintos lenguajes de programación.

Un lenguaje débilmente "tipado" no define de forma explícita el tipo de las variables (y por lo tanto las operaciones que son válidas con la variable), sino que el tipo de variable se determina durante la ejecución, en función de los valores que se les asignan. Matlab pertenece a este grupo de lenguajes:

```
a=3
a="Hola mundo"
```

Como puede observarse, la variable **a** almacena indistintamente un número entero (3) y una cadena de caracteres ("Hola mundo").

Por el contrario en un lenguaje fuertemente tipado, se determina de forma explícita el tipo de las variables (y de nuevo las operaciones que son válidas), el siguiente código no es válido en un lenguaje fuertemente tipado como C++ (nótese que ***int*** indica que la variable es un número entero):

```
int a=3;
a="Hola mundo"; // error no se puede asignar una cadena de caracteres a un entero directamente
```

1.4.6 Lenguajes que soportan la programación orientada a objetos

La programación orientada a objetos supone un cambio de mentalidad a la hora de programar ya que se pasa de trabajar con instrucciones y datos por separado a unirlos en lo que se denomina un objeto. De hecho el tipo de un objeto queda definido por las operaciones que pueden realizarse sobre él y no por los datos que contiene. Los tipos de los objetos se pueden definir en estos lenguajes mediante clases. La programación orientada a objetos permite el desarrollo de aplicaciones enormemente complejas de una forma más estructurada y robusta que mediante sólo la programación estructurada (que todos los lenguajes orientados a objetos soportan). Existe una gran cantidad de lenguajes que soportan este paradigma: C++, Java, C#, Simula, Smalltalk, Eiffel, Python, Ada, Lisp, OCaml, etc.

1.5 Errores

Antes de continuar, es importante diferenciar los dos tipos de errores que suelen aparecer con más frecuencia cuando se programa en cualquier lenguaje de programación.

1.5.1 Errores de sintaxis

Un error de sintaxis es aquel en el que el programa no sigue las reglas sintácticas definidas por el lenguaje. Por ejemplo la frase "El ratón comer un queso", no sigue las reglas de la lengua castellana, y por lo tanto es incorrecta. De igual forma, en un lenguaje de programación pueden escribirse programas incorrectos.

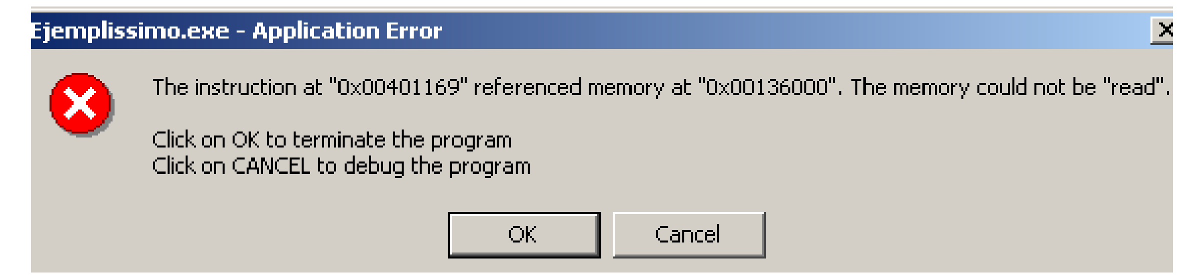
1.5.2 Errores lógicos

Estos errores suelen ser los más complejos de localizar, ya que el entorno de desarrollo no suele ser capaz de detectarlos hasta que el programa está en **funcionamiento**. En este caso el error no se produce a causa de que el programa no siga la sintaxis sino por un fallo en la lógica del programa que hace que su funcionamiento no sea el que desea el programador. Este tipo de fallos son semejantes a la frase "El queso come un ratón" aunque es sintácticamente correcta la lógica del sentido común nos dice que es imposible.

Ningún compilador o intérprete es capaz de localizar los errores lógicos. El error lógico se hace presente durante el tiempo de ejecución del programa. Los errores en tiempo de ejecución se detectan observando el programa, ya que éste no hace lo que el programador desea, y pueden dividirse en dos tipos:

Errores lógicos con aviso: Puede suceder que el error sea tal envergadura que haga saltar el mecanismo de protección del sistema operativo (en caso de programas compilados) o que el intérprete (en caso de programas interpretados) nos avise del error.

Al darse un error de este tipo en el caso de un programa compilado para trabajar bajo Windows, puede salir un aviso diciendo: "**This program has performed an illegal operation and will be shut down**" o mensajes como este:



Errores lógicos sin aviso: Son errores lógicos que pasan desapercibidos porque la única notoriedad que presentan es que el programa no hace lo que debería hacer.

Tanto para la localización y eliminación los errores lógicos con aviso como sin aviso, se requiere de una herramienta adicional denominada ***debugger***, que se ve a continuación.

1.5.3 Debugger

Un debugger es un programa con las siguientes capacidades:

- Permite ejecutar las líneas del programa una a una.
- Permite establecer puntos en el programa en los que el flujo del mismo debe detenerse pasando el control al debugger. Estos puntos son conocidos como puntos de ruptura, o más comúnmente mediante su nombre en inglés "breakpoint"
- Permite ver (y en algunos casos modificar) el contenido de las variables

El debugger permite comprobar como se ejecuta en la realidad un programa y supone una ayuda inestimable en el desarrollo de programas.

No hay otra alternativa que acostumbrarse a utilizarlo. El perfil de la persona que utilizar el debugger es el siguiente:

- Persona de edad variable que no le gusta perder mucho tiempo programando.
- Antes de programar lee atentamente las especificaciones.
- Dedica tiempo a pensar cómo hacer el programa y en cómo gestionar la información necesaria.
- Sabe lo que quiere programar aunque no lo haya conseguido.
- Sabe lo que tiene que hacer el programa.
- Tiene un papel, folio u hoja a su lado con el esquema del programa.

En cambio, el perfil de la persona que NO utiliza el debugger es este:

- Persona joven que se inicia a la programación y que dispone de mucho tiempo por delante para hacer un programa.
- No entiende lo que él mismo ha programado aunque insiste en escribir alguna que otra instrucción por ver si pasa algo.
- Tiende a mantener aquellas líneas de código que no dan errores y a borrar aquellas líneas de las que se queja el PC.
- No suele tener un papel o folio a su lado y, cuando lo tiene, éste se encuentra en blanco.

Capítulo 2 Estructuras fundamentales de los datos

2.1 Introducción

La información válida para un ordenador son números binarios. Los números suponen una simplificación de la realidad en el sentido de que solamente se guardan características particulares eliminando las otras que son irrelevantes.

Por ejemplo, cuando se realiza un programa para almacenar empleados de una empresa, los datos que se guardan de cada empleado son el DNI, la edad, el salario... pero no el color del pelo, el peso etc...

La elección de los datos a almacenar está condicionada por el problema a resolver. A veces esta determinación resulta evidente, a veces es difícil. Sin embargo el programa final que se obtiene -en términos de memoria que ocupa, rapidez de funcionamiento... - está muy condicionado por esta elección.

2.2 Las variables y las constantes

Una variable o una constante es un dato, o conjunto de datos, en la memoria del ordenador de especial interés para el programa y que recibe un nombre simbólico dentro del código del programa.

Cuando el valor de ese dato puede modificarse, se dice que es una variable, mientras que si no puede modificarse se dice que es una constante.

Tal y como se emplean dentro del código en la mayoría de los lenguajes de programación, las variables tienen las siguientes características (Observa la inicialización de la variable **a** para C++ en el ejemplo del apartado 1.4.5):

- Un nombre o identificador con el que nos referimos en el código a la variable. En el ejemplo indicado, el nombre es **a**.
- Un tipo, que indica los posibles valores que puede tomar la variable y las acciones que son posibles con ella. En el ejemplo, el tipo es un número entero (**int** es la expresión para definir un **integer** en C++).
- Un valor que está contenido en la memoria del ordenador. Inicialmente es el número 3.
- Una dirección de memoria que indica dónde está localizada la variable en la memoria (esta dirección la gestiona el compilador o el intérprete y puede ser o no ser accesible en el lenguaje de programación, como sucede en C++ o Matlab respectivamente).

Por su parte, las constates se utilizan como las variables, salvo que no se les puede aplicar la operación de asignación, ya que esta operación es para modificar el valor de la constante y, para evitar esta una contradicción, el compilador no lo permitirá. Las constantes se utilizan para almacenar valores constantes (valga la redundancia) como puede ser el número Pi, el número de segundos que tiene un minuto o la velocidad de la luz. A continuación sólo nos referiremos a las variables pero, salvando la comentada operación de asignación, las propiedades entre ambas son las mismas.

2.3 Tipos de datos

Internamente los ordenadores trabajan con una representación binaria (mediante unos y ceros) de los números. Esta forma de representar la información es muy adecuada a la hora de fabricar los circuitos electrónicos ya que:

1. Es la forma más básica de discernir.
2. Un 1 y un 0 son muy diferentes eléctricamente (tradicionalmente son 5 V de diferencia) por lo que es difícil que se confundan entre sí.

Sin embargo hacer un programa directamente con unos y ceros es prácticamente imposible. Preferimos utilizar palabras para las instrucciones y números en base 10. De ahí que los lenguajes de programación ofrezcan la posibilidad de utilizar distintos tipos variables para almacenar la distinta información.

A continuación se especifica la tipología de variables más comunes en los distintos lenguajes.

2.3.1 Variables booleanas

Una variable booleana sólo puede tener dos valores, verdadero o falso. Para representar una variable booleana sólo se necesita un bit (precisamente un bit es la cantidad mínima de información y sólo puede tener dos valores: 1 y 0), haciendo corresponder el valor 1 a verdadero y 0 falso. No obstante, hay que añadir que, en general, los computadores no son capaces de trabajar directamente con un solo bit, sino que trabajan con un paquete mínimo de 8 bit. Esta cantidad es conocida como byte.

Existe una serie de operadores que permiten trabajar con variables booleanas, de forma que se pueden agrupar y formar expresiones complejas. Los más habituales son los siguientes (empleando la sintaxis de Matlab):

Operador ***and*** (&):	1 & 1 = 1	1 & 0 = 0	0 & 1 = 0	0 & 0 = 0
Operador ***or*** (|):	1 | 1 = 1	1 | 0 = 1	0 | 1 = 1	0 | 0 = 0
Operador ***not*** (~):	~ 1 = 0	~ 0 = 1		

Estas relaciones se suelen escribir en tablas que muestran los posibles resultados de una de estas operaciones para todas las posibles entradas. Estas tablas reciben el nombre de tablas de verdad. La siguiente tabla muestra la tabla de verdad correspondiente al operador lógico AND y OR, respectivamente:

AND	**verdadero**	**falso**
verdadero	verdadero	falso
falso	falso	falso

El resultado de una operación AND es verdadero si los dos operandos son verdaderos.

OR	**verdadero**	**falso**
verdadero	verdadero	verdadero
falso	verdadero	falso

El resultado de una operación OR es verdadero si, al menos, uno de los dos operandos es verdadero.

2.3.2 Números enteros

Los números enteros suelen emplearse con gran frecuencia en programación, ya que su procesamiento es muy rápido y existen muchas operaciones en las que son imprescindibles como, por ejemplo:

- Como los indices en un vector o matriz
- Como contador del número de iteraciones en un bucle

Debido a limitaciones de memoria no se puede definir un número con una precisión arbitraria.

Los lenguajes de programación pueden definir distintos tamaños para los enteros; empleando, por ejemplo, 8 bits (para contar desde el 0 hasta el 256), 16 bits (del 0 al 65536 o -32768 a 32761 según se considere el signo o no), etc.

Como ejemplo con 4 bits y dedicando el primer bit para el signo, una posible representación binaria es:

Decimal	Binario	Decimal	Binario
0	0000	-1	1111
1	0001	-2	1110
2	0010	-3	1101
3	0011	-4	1100
4	0100	-5	1011
5	0101	-6	1010
6	0110	-7	1001
7	0111	-8	1000

Se puede comprobar que el mayor número positivo es 2^3-1 y el menor número negativo es -2^3. O lo que es lo mismo:

$$MAX = 2^{nbits-signo} - 1$$

$$MIN = -2^{nbits-signo}$$

Donde *nbits* es el número de bits de la variable, y *signo* es 1 si la variable tiene signo o 0 en caso de que sólo se consideren números positivos.

Con los números enteros se pueden hacer operaciones matemáticas. Los típicos símbolos empleados en los lenguajes de programación son:

- **+** para sumar
- **-** para restar
- ***** para multiplicar
- **/** para dividir

Pero hay que notar que en general la división de dos números enteros es otro número entero y por lo tanto sin decimales, esto es, se produce un truncamiento del resultado. Por ejemplo, para el siguiente programa en C++:

```
#include <iostream.h>
void main(void){
int a=2;
int b=5;
int c=a/b;
cout<<"C: "<<c<<endl;
}
```

Resulta que el contenido de la variable **c** es 0.

2.3.3 Números de coma flotante

Los ordenadores trabajan también con números decimales. En este caso el ordenador trabaja con un número máximo de cifras significativas. La cifras significativas de los números 0.0001562, -1341 y 23740000 son cuatro. Se puede escribir estos números de forma científica de la siguiente forma: $0.1562*10^{-3}$, $-0.1341*10^{+4}$ y $0.2374*10^{+8}$. Los ordenadores codifican los números decimales de una forma muy parecida.

Por motivos de eficiencia se suelen distinguir dos tipos diferentes: los números de precisión simple (32 bits y 6 o 7 cifras significativas) y los de precisión doble (64 bits y 15 cifras significativas).

Con los números en coma flotante se pueden hacer las mismas operaciones matemáticas que con los números enteros, pero en este caso, el ordenador considerará los decimales.

```
#include <iostream.h>
void main(void){
float a=2;
float b=5;
float c=a/b;
cout<<"C: "<<c<<endl;
}
```

En este caso el valor de **c** es 0.4.

2.3.4 Caracteres

Al igual que los números los caracteres se codifican de forma binaria. A cada letra del alfabeto se le asigna un determinado valor que la representa. Evidentemente existen diferentes formas de codificar las letras. Destacan por su importancia:

- ASCII, emplea 7 bit para representar las letras del alfabeto latino, números y símbolos propios de la lengua inglesa. Por ejemplo, el carácter ASCII de la letra **a** es 97.
- ANSI, representa las letras mediante un byte (8 bit). La codificación ANSI extendió la codificación ASCII para incluir las letras acentuadas y especiales de idiomas como el francés, el alemán o el castellano (el cual tiene la Ñ).
- UNICODE UTF-16, emplea 2 bytes para representar cualquier carácter en cualquier idioma, lo que incluye el árabe, hebreo, chino, japonés… Por ejemplo la letra hebrea alef א tiene un código UTF-16 de 1488.

- UNICODE UTF-8, es un intermedio entre los dos anteriores. No emplea un número de bytes fijo para representar los caracteres, pero está diseñado de tal forma que los textos escritos mediante esta codificación sólo empleen letras del alfabeto latino con el código ASCII. Por motivos que no se explican aquí, ya que se escapan del interés de este manual, esta codificación será una de las más empleadas en el futuro junto a la UTF-16.

2.4 Datos estructurados

Hasta ahora se han visto tipos de variables que representan una entidad indivisible o atómica. No obstante en ocasiones puede ser preferible trabajar con variables de una forma agrupada. Por ejemplo, para programar algo para trabajar con números complejos, es mejor crear un nuevo tipo de variable para representar los números complejos, donde quede una variable de este tipo definida por su parte real y su parte imaginaria (dos números de coma flotante). Así mismo, en los programas donde se manipulan muchos datos de la misma naturaleza es más sencillo trabajar con una variable que agrupe a su vez a varias variables del mismo tipo en un solo vector.

2.4.1 Vectores y cadenas de caracteres

Un vector es un tipo de variable compuesto por N variables del mismo tipo, por ejemplo un vector de 10 enteros o de 15 números de coma flotante.

Por lo tanto las características de un vector son:

- El tipo de los elementos que lo conforman
- El número de elementos

El acceso a los elementos del vector se realiza siempre mediante un índice que es siempre un número entero positivo. El índice del primer elemento de un vector puede ser 1 o 0 según el lenguaje de programación siendo más común el segundo.

El siguiente ejemplo muestra la creación de un vector de números decimales en Matlab y el acceso al segundo elemento (en Matlab los índices de un vector comienzan con 1):

```
>> a=[3,5,9,17]
a =
   3    5    9   17
>> a(2)
ans =
   5
```

Las cadenas de caracteres son un tipo especial de vector en los que cada elemento es un carácter. Este tipo de variable se suele emplear para almacenar cualquier tipo de texto. En algunos lenguajes de programación (en especial C++ y C) emplean (internamente) un carácter especial de código para marcar el fin de la cadena de caracteres. El siguiente ejemplo es similar al anterior pero emplea cadenas de caracteres:

```
>> text='Hola mundo'
text =
Hola mundo
>> text(4)
ans =
a
```

2.4.2 Matrices

Las matrices se distinguen de los vectores en que la disposición de los elementos no es unidimensional sino bidimensional. Las matrices tienen las mismas características que los vectores salvo que para acceder a los elementos se requieren dos índices en lugar de uno. Las matrices son ampliamente empleadas tanto para representar tablas de datos, rotaciones en el espacio, o incluso imágenes. Se pueden crear matrices de más de dos dimensiones, conocidas como hipermatrices.

2.4.3 Estructuras

Todos los elementos de un vector o de una matriz son iguales. Por ejemplo, en un vector de enteros, todos y cada uno de los elementos del vector son enteros. Sin embargo es muy útil poder trabajar con variables compuestas por distintos tipos de elementos. Por ejemplo para almacenar los datos de los empleados de una empresa se pueden crear vectores que almacenen por separado el nombre, edad, número de la seguridad social, etc. aunque es mucho más cómodo poder trabajar con un nuevo tipo de variable que permitiese agruparlos en una única entidad y almacenarlos en un único vector, evitando que aparezcan incoherencias en los datos (por ejemplo si se ordena el vector de nombres de los empleados, hay que trasladar sincronizadamente el resto de datos de los empleados para que sigan refiriéndose a la persona a la que pertenecen).

Una estructura es una agrupación de datos de distinto o igual tipo que representan de algún modo una única entidad. Los componentes de una estructura suelen recibir el nombre de **miembros** de la estructura y **nunca** son accesibles a través de un índice.

El siguiente ejemplo muestra la apariencia de una estructura en C++:

```
struct Empleado
{
	char nombre[10];	// Una cadena de caracteres de tamaño 10
	int  edad;		// Un entero
	int  seguridadSocial;
};
```

Aunque Matlab permite la creación de estructuras, su uso es menos natural que en otros lenguajes de programación.

Los lenguajes de programación orientada a objetos como C++, añaden a la capacidad de crear variables la capacidad de añadir funciones miembro a las estructuras, que en este caso reciben el nombre de clases. Las funciones miembro suelen definir la capacidad (lo que son y lo que pueden hacer) de las variables del tipo que define una clase y reciben el nombre de interfaz de la clase. Así mismo estos lenguajes permiten impedir el acceso a los datos desde su exterior de forma que se puede garantizar la consistencia de los datos, así como ampliar la interfaz de una clase mediante un mecanismo denominado herencia.

Capítulo 3 El flujo de un programa

3.1 Introducción

Para cualquier tarea que desee realizarse es necesario, en primer lugar, entender qué ha que hacer. En este sentido, la inteligencia humana es capaz de interpretar instrucciones vagas o incompletas y hacerse cargo de qué hay que hacer realmente, observando el contexto.

Una vez entendida la tarea, hay que establecer los pasos a seguir para llegar al objetivo propuesto. Un algoritmo es un conjunto de pasos que al ser seguidos se consigue realizar una tarea o resolver un problema. Para realizar algoritmos fácilmente entendibles los programadores utilizan diagramas de flujo. Los diagramas de flujo son diagramas para expresar los pasos de un algoritmo por medio de símbolos conectados por líneas. Es como un "mapa" donde aparecen simultáneamente:

- Las rutas que puede seguir el flujo de datos al ejecutar un algoritmo. En algún punto es posible que el camino se divida en varios, en ese caso, el diagrama indica que si se cumplen unas determinadas condiciones, se escogerá un camino, si se cumplen otras, se escogerá otro, etc.
- Las acciones y operaciones que hay que realizar en puntos concretos del camino que se recorre.

3.2 El origen de los diagramas de flujo

Herman H. Goldstine nos lo cuenta:

"In the spring of 1946, that year John von Neumann and I evolved an exceedingly crude sort of geometry drawing to indicate in rough fashion the iterative nature of an induction. At first this was intended as a sort of tentative aid to us in programming. Then that summer I became convinced that this type of flow diagram, as we named it, could be used as a logically complete and precise notation for expressing a mathematical problem and that indeed this was essential to the task of programming. Accordingly, I developed a first, incomplete version and began work on the paper called Planning and Coding... Von Neumann and I worked on this material with valuable help form Arthur Walter Burks and my wife (Adele Katz Goldstine). Out of this was to grow not just a geometrical notation but a carefully thought out analysis of programming as a discipline. This was done in part by thinking this though logically, but also and perhaps more importantly by coding a large number of problems. Through this procedure real difficulties emerged and helped illustrate general problems that were then solved.

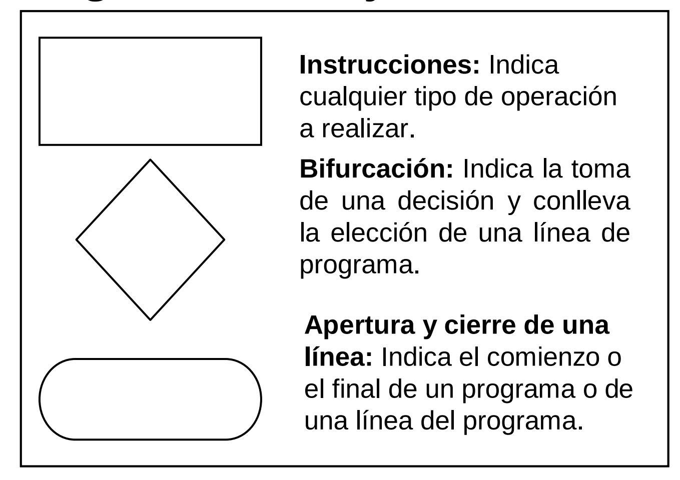
The purpose of the flow diagram is to give a picture of the motion of the control organ as it moves through the memory picking up and executing the instructions it finds there. The flow diagram also shows the states of the variables at various key points in the course of the computation. Further, it indicates the formulas being evaluated...."[2]

[2] Herman H. Goldstine, *The computer from Pascal to von Neumann*. Princeton, N.J.: Princeton University Press, 1972, 1980, pp. 266-267.

3.3 Elementos de un diagrama de flujo

A continuación se presentan los 3 principales símbolos de un diagrama de flujo que se utilizarán en este manual y en la asignatura de Informática I y Fundamentos de Computadores.

Las líneas de los diagramas de flujo que unen los símbolos indican el camino que lleva el flujo del programa.



Los símbolos rectangulares contienen instrucciones, acciones que el programa debe realizar una vez llegado a ese punto.

Los símbolos con forma de rombo son puntos en los que el camino a seguir se divide en dos. Dentro de este símbolo hay una pregunta, que sólo puede dos respuestas posibles: “Sí” o “No”. Se tomará un camino u otro dependiendo de la respuesta a la pregunta del interior del símbolo.

El tercer tipo de símbolos indican que se comienza el proceso, o bien que éste se finaliza. Representan el comienzo o el final de un programa, o de una línea del programa.

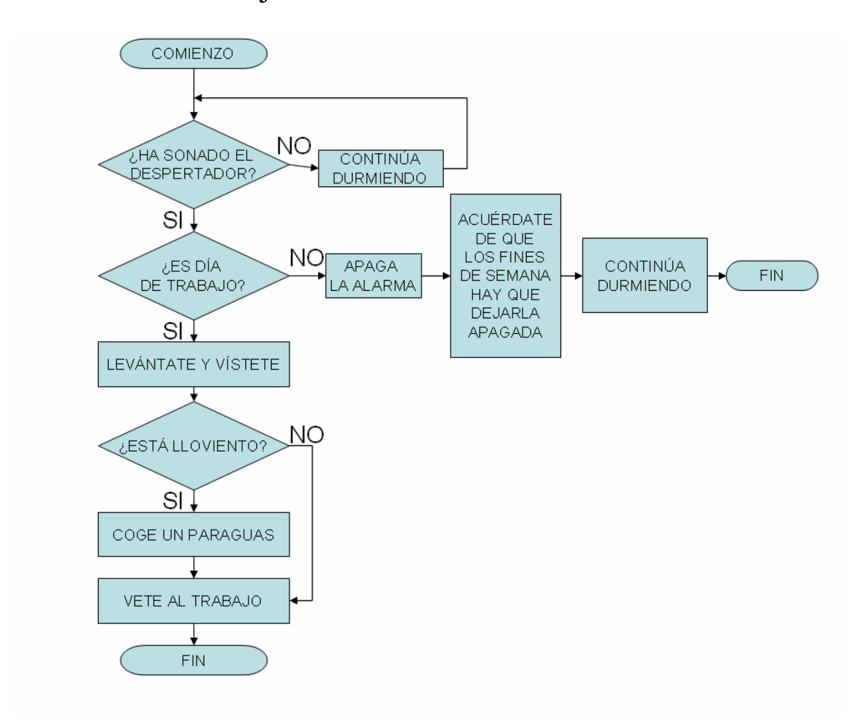
3.4 Desarrollo de un diagrama de flujo para un proceso cotidiano

Casi cualquier proceso se puede representar mediante un diagrama de flujo, también los procesos cotidianos. Cuando se planea algo, ese plan también se puede representar mediante un diagrama en el que se describen los pasos a dar y las decisiones a tomar dependiendo de las condiciones que se estén dando en cada etapa.

La siguiente figura representa un ejemplo de cómo se podría representar un proceso de la vida cotidiana, el hecho de levantarse una mañana cualquiera e ir a trabajar, mediante un diagrama de flujo. Al fin y al cabo, para mucha gente levantarse e ir a trabajar es como ejecutar un programa de ordenador, con sus rutinas y subrutinas…

El símbolo “COMIENZO” indica el estado inicial del proceso, el punto de partida: el individuo se encuentra en su cama, durmiendo placenteramente. Al comenzar a ejecutarse el programa lo primero que se hace es realizar un ***Chequeo***, en el que se comprueba si ha sonado el despertador o no. En caso negativo, se ejecuta la ***Instrucción*** de continuar durmiendo y se regresa al estado inicial del proceso. En caso afirmativo, hay que hacer un nuevo ***Chequeo***: ¿es hoy día de trabajo? Si no lo es, hay que seguir el camino indicado por el diagrama, que consta de varias ***Instrucciones***: apagar la alarma, acordarse de que dejar la alarma encendida un fin de semana es un grave error, y seguir durmiendo. Una vez llegado a este punto se habría acabado el proceso de levantarse e ir al trabajo, se llega a un símbolo de “FIN”. En este caso concreto no se habría realizado el proceso de levantarse e ir a trabajar, porque tras los sucesivos chequeos se ha comprobado que no se cumplían las condiciones necesarias para ello.

Pero volvamos al punto en el que se chequea si es día de trabajo. En caso de que sí lo sea, hay que continuar por el camino indicado. Hay que obedecer a la ***Instrucción*** de levantarse y vestirse. Tras hacerlo, parece que se está en condiciones de continuar con el proceso, pero antes (sobre todo en una ciudad como San Sebastián), hay que ***Chequear*** si está lloviendo o no. Si no llueve, el proceso se puede hacer más rápido, directamente habrá que seguir la ***Instrucción*** de ir a trabajar. En caso contrario, hay que obedecer una ***Instrucción*** previa, que es la de coger un paraguas, antes de irse al trabajo. Una vez cumplidas estas instrucciones, termina el proceso representado por el diagrama, se llega al símbolo de "FIN", que indica el estado final, en el que el individuo se encuentra de camino al trabajo.



En este caso, el proceso tiene dos estados finales. En un diagrama de flujo puede haber un único estado final, o varios. Partiendo de un estado inicial, y dependiendo de las condiciones y circunstancias que se vayan dando al ir completando el proceso, que vendrán determinadas en los sucesivos ***Chequeos***, se podrá acabar en uno u otro estado final, incluso siguiendo caminos diferentes para llegar al mismo estado.

3.5 Instrucciones

El primer tipo de elementos que se han descrito en el apartado anterior son las ***Instrucciones***. Como la propia palabra indica, se trata de una serie de acciones que se han de realizar en ese punto del recorrido representado por el diagrama. También es frecuente llamarlas ***sentencias***.

Mediante las instrucciones o sentencias se manipulan los datos que el lenguaje de programación conoce: se realizan cálculos, se asignan valores, etc. Los datos se manejan mediante los operadores aritméticos, los operadores lógicos y los procedimientos y funciones. En sucesivos capítulos se verá con detalle qué son y cómo trabajan.

Una vez que se han ejecutado todas las instrucciones de uno de los símbolos del diagrama, se está en condiciones de continuar el camino, bien hacia un nuevo bloque de instrucciones, bien hacia un nuevo ***Chequeo***, o bien hasta el final del programa.

3.6 Operaciones aritmético lógicas

En la gran mayoría de los lenguajes de programación se utilizan "operadores" para representar operaciones aritméticas. Por ejemplo, el operador + hace que se sumen los valores situados a su izquierda y su derecha. Aunque siempre hay algún lenguaje extraño por ahí que emplea algún operador diferente, casi todos los lenguajes de programación emplean los operadores de manera muy similar. Se distinguen 4 tipos fundamentales de operadores: los matemáticos, los lógicos, los relacionales y el de asignación.

3.6.1 El operador de asignación

En la mayoría de los lenguajes de programación el signo "=" **no** significa "igual a". Es un operador de **asignación** de valores, y no hay que confundirlo con el operador relacional de igualdad (==). Veamos algunos ejemplos de asignación:

```
a = 6;
string = 'Paco el bombero';
resultado = a – 1;
resultado = resultado / 2;
```

La primera sentencia asigna el valor numérico 6 a una variable llamada **a**.

La segunda sentencia hace que una variable llamada **string** almacene una cadena de caracteres que contiene las palabras "Paco el bombero".

La tercera sentencia hace que en una variable llamada **resultado** se almacene el valor numérico que tiene la variable **a** menos el valor numérico 1, es decir, en estre caso la variable **resultado** almacenaría el valor 5.

La cuarta sentencia divide entre 2 el valor que tenía la variable **resultado**, es decir, después de ejecutarse en este caso, **resultado** pasaría a valer 2,5. Hay que notar que desde el punto de vista matemático la expresión no tiene sentido. Ya se ha dicho que no es un operador de igualdad, sino de sustitución. Por eso, la sentencia

```
a + b = 5;
```

no es válida. A la izquierda del operador "=" no puede haber nunca una expresión. Tiene que haber necesariamente una variable. De esta forma tampoco es válida la expresión

```
a + b = 5 + c = d-f;
```

Ni cualquiera de sus derivados.

3.6.2 Operadores aritméticos o matemáticos

Los operadores matemáticos permiten realizar operaciones aritméticas básicas con los tipos de datos que intervienen en un programa. Como es lógico, los datos que pueden manejarse con estos operadores han de ser numéricos: bien números propiamente dichos, o variables cuyo contenido sea numérico, o bien valores de retorno de funciones que sean numéricos (más adelante se explicará qué es un valor de retorno de una función).

Es importante saber con qué prioridad el programa va a efectuar los cálculos cuando se tienen varios operadores en una misma sentencia. En el caso de C y Matlab, hay que tener claro que en primer lugar se resuelven los paréntesis, después las multiplicaciones y divisiones y, en tercer lugar, las sumas y restas. La expresión

resultado = numero*5/4+3;

contiene operandos y operadores. En este caso, el valor de la variable **resultado** se calcularía del siguiente modo:

1º resultado = numero * 5

2º resultado = resultado / 4

3º resultado = resultado + 3

La mejor forma de evitar problemas por la prioridad de operadores es indicar mediante paréntesis cómo se deben ejecutar las operaciones. En este ejemplo:

resultado = (numero*(5/4)) + 3;

Si se quiere forzar un orden de prioridad diferente, también se han de emplear los paréntesis, de forma parecida a las expresiones matemáticas. La expresión

resultado = numero*5/(4+3);

asigna a la variable **resultado** el valor de la variable **numero** multiplicado por 5/7.

3.6.3 Operadores relacionales

Este es un apartado especialmente importante para todas aquellas personas sin experiencia en programación. Una característica imprescindible de cualquier lenguaje de programación es la de considerar alternativas, esto es, la de proceder de un modo u otro según se cumplan o no ciertas condiciones. Los operadores relacionales permiten estudiar si se cumplen o no esas condiciones. Así pues, estos operadores producen un resultado u otro según se cumplan o no algunas condiciones que se verán a continuación.

En el lenguaje natural, existen varias palabras o formas de indicar si se cumple o no una determinada condición: sí o no, verdadero o falso (*true* o *false* en inglés), etc.

En la mayoría de lenguajes de programación se ha hecho bastante general el utilizar la última de las formas citadas: (*true*, *false*). Si una condición se cumple, el resultado es **true** (se considera un "sí"); en caso contrario, el resultado es **false** (se considera un "no"). Lo más frecuente es que un 0 representa la condición de **false**, y cualquier número distinto de 0 equivale a la condición **true**. Cuando el resultado de una expresión es **true** y hay que asignar un valor concreto distinto de cero, por defecto se toma un valor unidad. Los operadores relacionales son los siguientes:

Menor que <

Mayor que >

Menor o igual que <=

Mayor o igual que >=

Igual que ==

Distinto que ~=

Los operadores relacionales son operadores binarios, es decir, tienen dos operandos. Su expresión general es

expresion_1 operador expresion_2

donde operador es cualquiera de los vistos (<, >, <=, >=, == ó ~=). El resultado de la operación será un 0 si la condición representada por el operador relacional no se cumple, y será un 1 si la condición representada por el operador relacional se cumple. Veamos algunos ejemplos:

(2==1)	el resultado es 0 porque la condición no se cumple
(3<=3)	el resultado es 1 porque la condición se cumple
(3<3)	el resultado es 0 porque la condición no se cumple
(1~=1)	el resultado es 0 porque la condición no se cumple

3.6.4 Operadores lógicos

Los operadores lógicos son operadores que permiten combinar los resultados de otros operadores, sobre todo de los operadores relacionales, comprobando que se cumplen simultáneamente varias condiciones, que se cumple una u otra, etc. Existen tres operadores lógicos: el operador "Y" (&), el operador "O" (|), y el operador "No" (~). En inglés son los operadores And y Or y Not. Su forma general es la siguiente:

expresion1 & expresion2
expresion1 | expresion2
~expresion1

El operador & devuelve un 1 si tanto **expresion1** como **expresion2** son verdaderas (o distintas de 0), y 0 en caso contrario, es decir si una de las dos expresiones o las dos son falsas (iguales a 0); por otra parte, el operador | devuelve 1 si al menos una de las expresiones es cierta.

El operador de negación ~ es unitario. Su resultado es la **expresion1** negada, es decir, si **expresion1** es **true**, el resultado de **~expresion1** será **false**, y viceversa. Por lo tanto, devolverá un 0 si **expresion1** es distinto de 0 y devolverá un 1 si **expresion1** es un 0.

Es importante tener en cuenta que muchos compiladores de lenguajes de programación tratan de optimizar la ejecución de estas expresiones, lo cual da bastante juego en ciertos casos. Por ejemplo: para que el resultado del operador & sea verdadero, ambas expresiones tienen que ser verdaderas; si se evalúa **expresion1** y es falsa, ya no hace falta evaluar **expresion2**, y de hecho no se evalúa. Algo parecido pasa con el operador | : si **expresion1** es verdadera, ya no hace falta evaluar **expresion2**.

Los operadores & y | se pueden combinar entre sí y agruparlos mediante paréntesis. Por ejemplo:

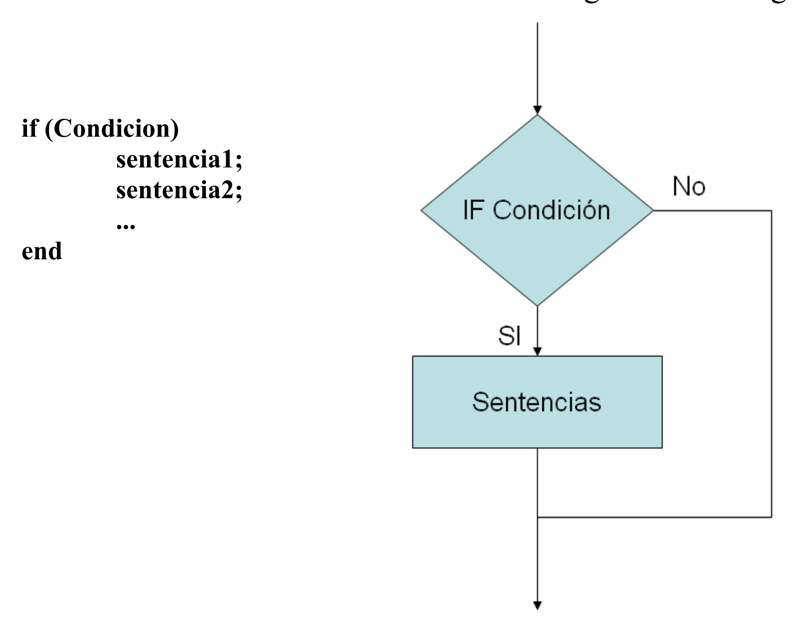
(2==1) | (-1==-1)	el resultado es 1
(2==1) & (-1==-1)	el resultado es 0
((2==2) & (3==3)) | (4==0)	el resultado es 1
((6==6) | (8==0)) & ((5==5) & (3==2))	el resultado es 0

3.7 Bifurcaciones

Las bifurcaciones se corresponden con lo que en los diagramas de flujo se han llamado ***Chequeos***. El flujo de datos del programa seguirá un camino u otro del diagrama según se cumpla o no la condición representada en el chequeo. La manera más habitual de encontrar las bifurcaciones es en forma de sentencias **if**, sentencias **if ... else**, o sentencias **if ... else** múltiples.

3.7.1 Sentencia *if*

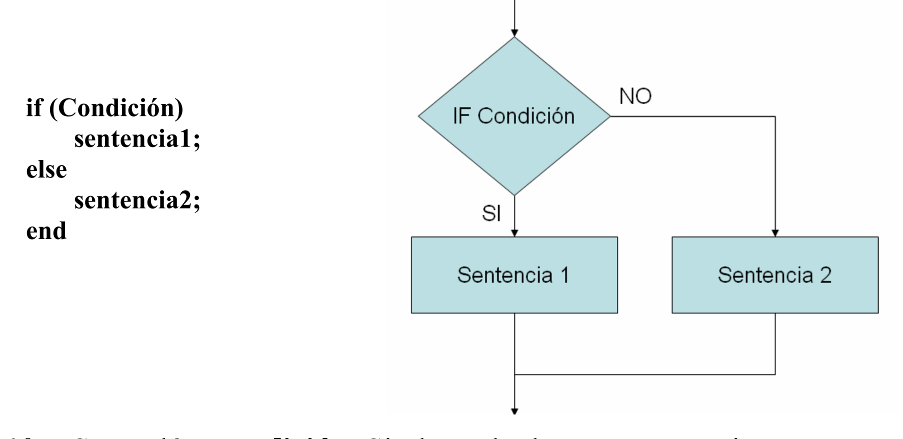
Esta sentencia de control permite ejecutar o no una sentencia simple o compuesta según se cumpla o no una determinada condición. Tiene la siguiente forma general:



Explicación: Se evalúa la '**Condicion**'. Si el resultado es **true**, se ejecutan las ***sentencias***; si el resultado es **false**, se saltan las ***sentencias*** y el programa prosigue.

3.7.2 Sentencia *if ... else*

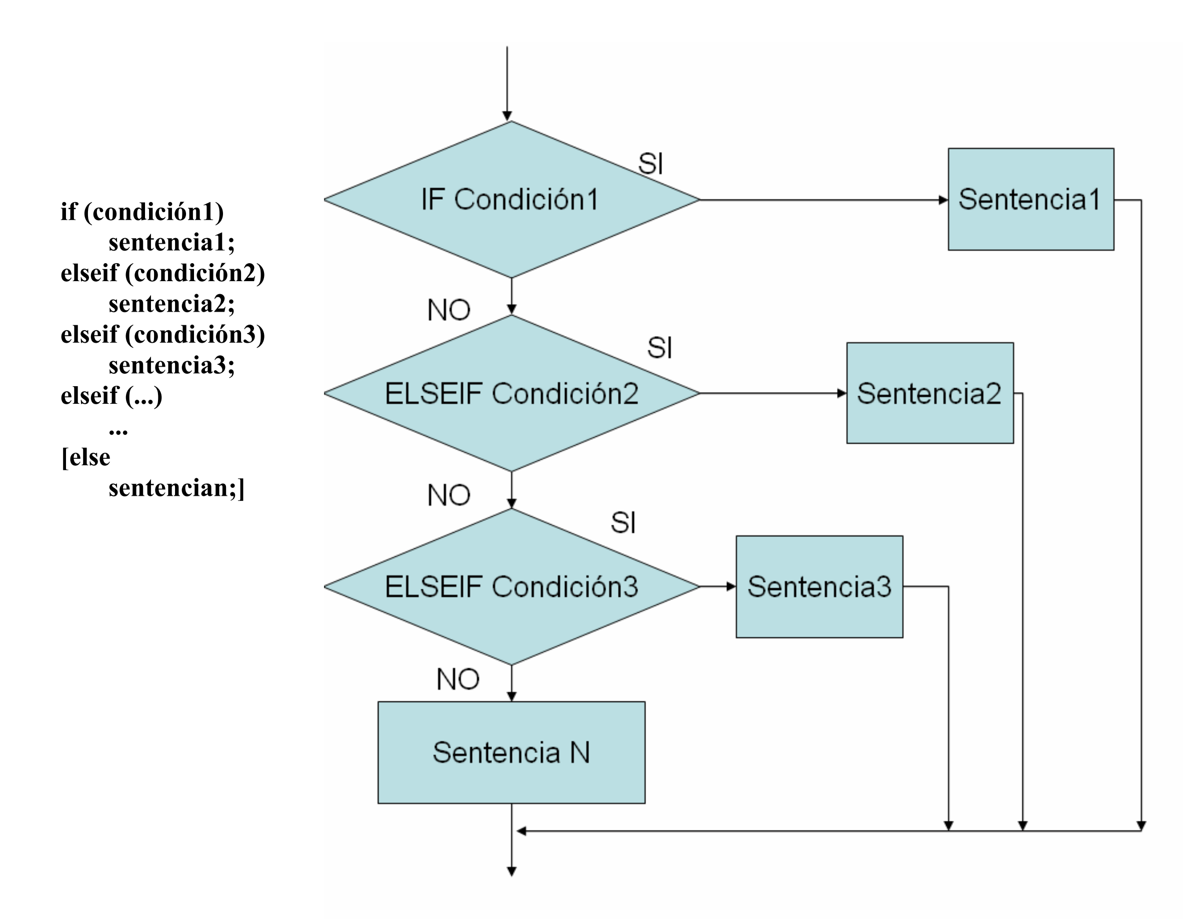
Esta sentencia permite realizar una bifurcación, ejecutando una parte u otra del programa según se cumpla o no una cierta condición. La forma general es la siguiente:



Explicación: Se evalúa ***Condición***. Si el resultado es **true**, se ejecuta **sentencia1** y se prosigue en la línea siguiente a **sentencia2**; si el resultado es **false**, se salta **sentencia1**, se ejecuta **sentencia2** y se prosigue en la línea siguiente.

3.7.3 Sentencia *if elseif... else* múltiple

Esta sentencia permite realizar una ramificación múltiple (en realidad son bifurcaciones sucesivas), ejecutando una entre varias partes del programa según se cumpla una entre **n** condiciones. La forma general es la siguiente:

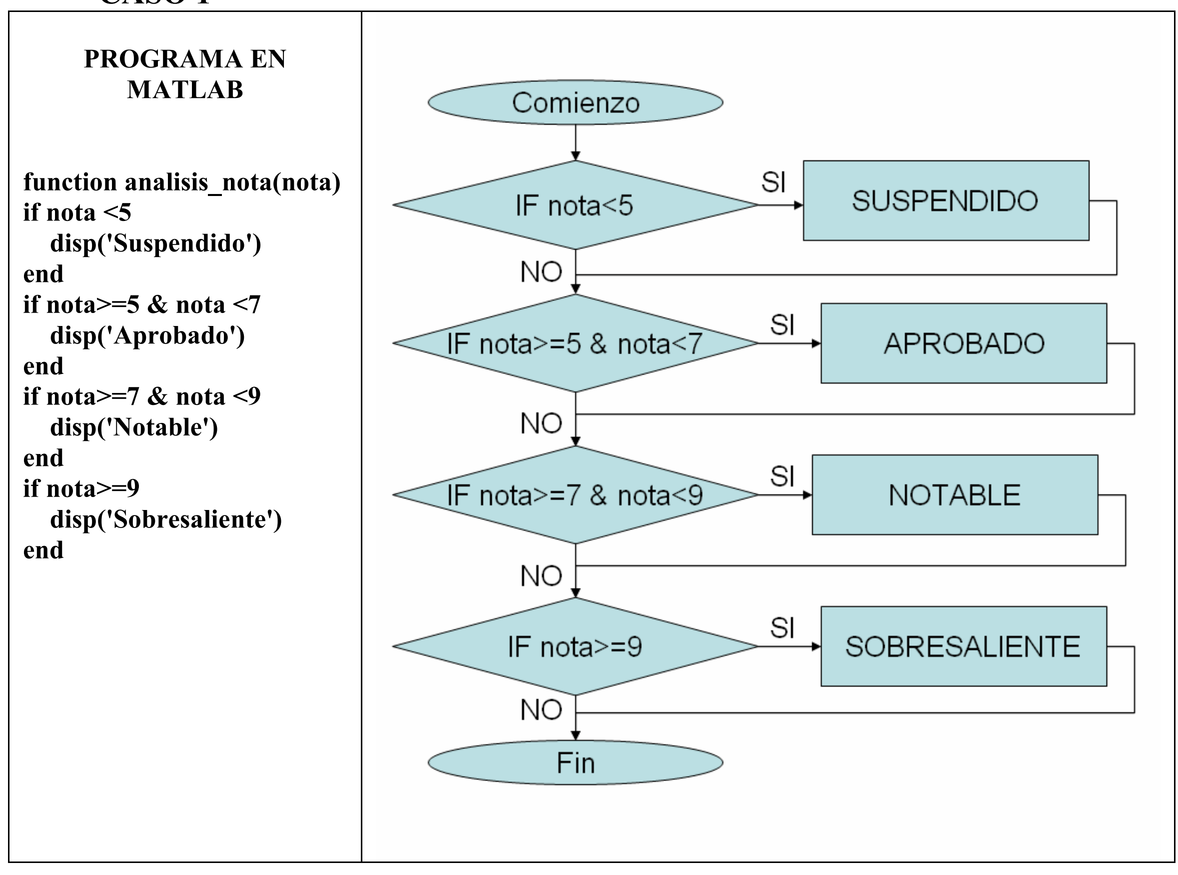


Explicación: Se evalúa condición1. Si el resultado es **true**, se ejecuta sentencia1. Si el resultado es **false**, se salta sentencia1 y se evalúa condición2. Si el resultado es **true** se ejecuta sentencia2, mientras que si es **false** se evalúa condición3 y así sucesivamente. Si ninguna de las condiciones o condiciones es **true** se ejecuta condiciónn que es la opción por defecto (puede ser la sentencia vacía, y en ese caso podría eliminarse junto con la palabra **else**).

El efecto conseguido con una bifurcación **if-elseif-else** puede conseguirse con una sucesión de bifurcaciones **if.** La ventaja de utilizar la estructura **if-elseif-else** está en que es más eficiente y más clara. Observa el siguiente ejemplo en el que surge la casuística planteada. Ambos programas sacan por pantalla la calificación que se corresponde (desde suspenso hasta sobresaliente) a una determinada nota (desde 0 hasta el 10).

El caso se plantea una sucesión de bifurcaciones IF, mientras que en el segundo se plantea una estructura IF-ELSEIF-ELSE.

CASO 1



PROGRAMA EN MATLAB

```
function analisis_nota(nota)
if nota <5
   disp('Suspendido')
end
if nota>=5 & nota <7
   disp('Aprobado')
end
if nota>=7 & nota <9
   disp('Notable')
end
if nota>=9
   disp('Sobresaliente')
end
```


En el ejemplo presentado arriba, el esquema que se ha seguido es chequear el intervalo en el que se encuentra la nota haciéndose las siguientes preguntas:

1º ¿Es inferior a 5?

2º ¿Está entre el 5 y el 7?

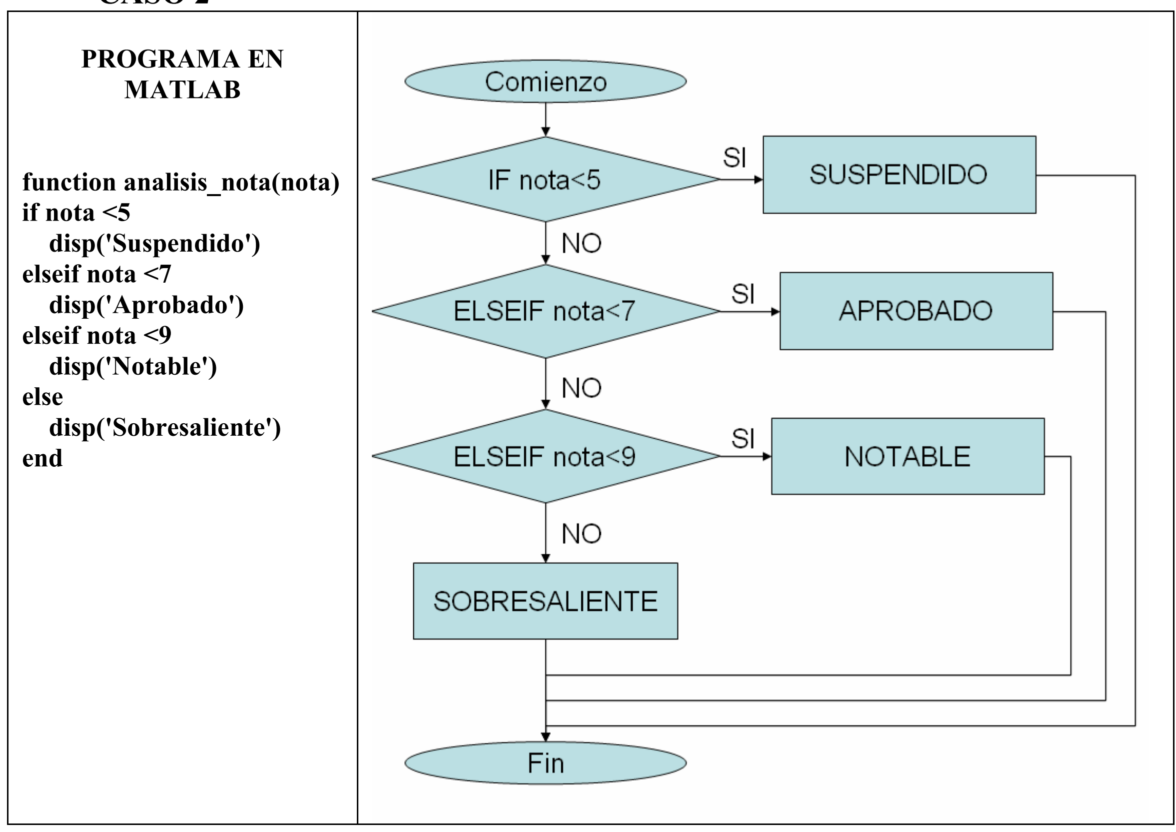
3º ¿Está entre el 7 y el 9?

4º ¿Es superior a 9?

Para realizar este chequeo se han introducido cuatro bifurcaciones IF consecutivas. Cada vez que se ejecuta el programa se hacen las 4 comprobaciones. Sin embargo, sabemos que una y sólo una de las cuatro comprobaciones se va a cumplir. Además, si la respuesta a la primera pregunta es NO, sería interesante poder utilizar esa información y no tener que chequear si la nota es mayor o igual a 5, ya que si la respuesta anterior ha sido NO, la nota es necesariamente mayor o igual a 5. Por último, si resulta que la nota es, por ejemplo, un 4, el resultado del primer chequeo es SI y no tiene sentido seguir realizando más comprobaciones, ya que podemos estar seguros que las otras tres comprobaciones restantes no se van a cumplir.

He aquí el interés de las estructuras IF-ELSEIF-ELSE. Hay muchas bifurcaciones en las que si se cumple una condición, el resto de condiciones no se van a cumplir, como el ejemplo propuesto. Así que el programa podría haberse concretado de la siguiente manera:

CASO 2


PROGRAMA EN MATLAB

```
function analisis_nota(nota)
if nota <5
   disp('Suspendido')
elseif nota <7
   disp('Aprobado')
elseif nota <9
   disp('Notable')
else
   disp('Sobresaliente')
end
```


Observa las líneas del flujo del programa. Al ser una estructura IF-ELSEIF-ELSE, al cumplirse alguna de las condiciones, se determina la nota y directamente se pasa al final del programa; esto es, no se continúa realizando comprobaciones.

Solamente si la nota es mayor o igual a 9 se realizarán todas las comprobaciones. Aún así, el chequeo de estas comprobaciones está más depurado que en el caso de la secuencia de bifurcaciones IF, ya que se aprovecha la información extraída de los anteriores chequeos.

De forma que ambos programas funcionarán correctamente, pero **es el segundo programa quien presenta una eficiencia superior**. Cuando se trabaja con programas cortos, la eficiencia es un factor de poca importancia, pero cuando se programa cosas más complicadas la eficiencia es un factor de gran importancia.

3.8 Bucles

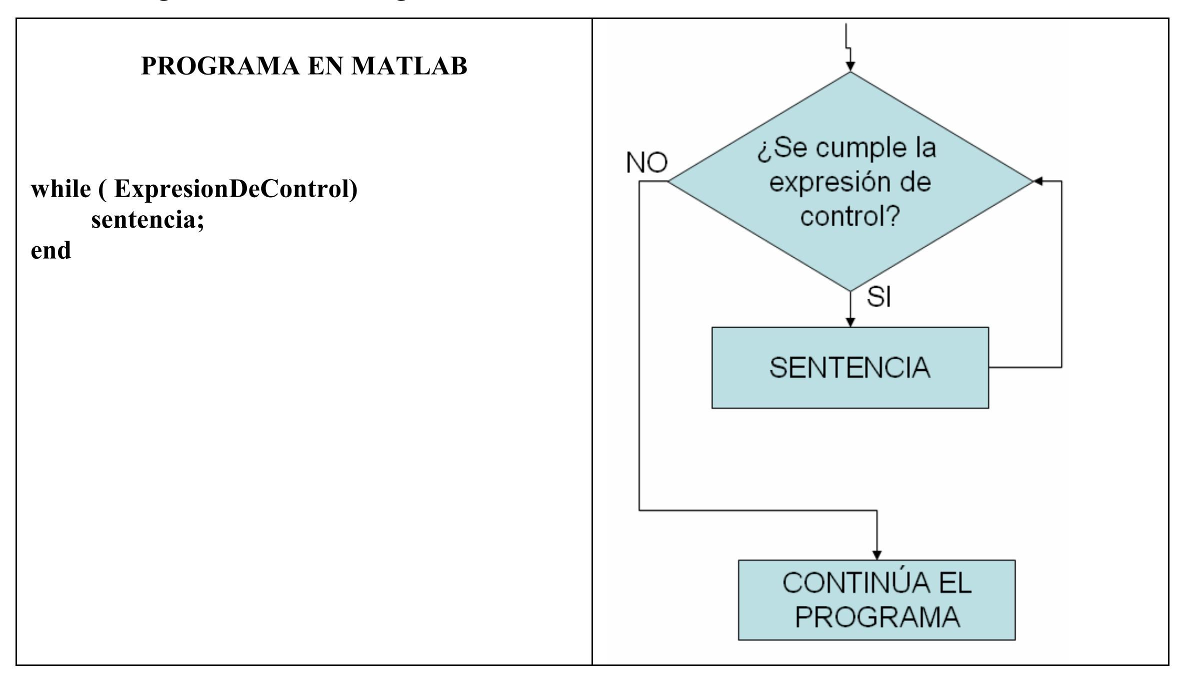
La otra gran herramienta en los lenguajes de programación para controlar el flujo de un programa son los bucles. Éstos permiten repetir la ejecución de líneas de código, es decir, permiten que el flujo del programa vuelva de nuevo hacia atrás. Esta repetición se realiza, bien un número determinado de veces, bien hasta que se cumpla una determinada condición de tipo lógico o aritmético. Aunque cada lenguaje de programación tiene sus propias construcciones, y su manera de utilizarlas, las dos construcciones más habituales para realizar bucles son el **while** y el **for**.

3.8.1 Bucle While

Este bucle ejecuta repetidamente una sentencia o bloque de sentencias mientras se cumpla una determinada condición. **Los bucles WHILE se utilizan cuando queremos**

que se repita una serie de instrucciones mientras que se cumpla una determinada condición que no sabemos cuándo dejará de cumplirse.

La forma general es como sigue:



Explicación: Se evalúa **ExpresionDeControl** y si el resultado es **false** se salta la **sentencia** y se prosigue la ejecución. Si el resultado es **true** se ejecuta **sentencia** y se vuelve a evaluar **ExpresionDeControl** (evidentemente alguna variable de las que intervienen en **ExpresionDeControl** habrá tenido que ser modificada, pues si no el bucle continuaría indefinidamente). La ejecución de **sentencia** prosigue hasta que **ExpresionDeControl** se hace **false**, en cuyo caso la ejecución continúa en la línea siguiente a sentencia. En otras palabras, **sentencia** se ejecuta repetidamente mientras **ExpresionDeControl** sea **true**, y se deja de ejecutar cuando **ExpresionDeControl** se hace **false**. Obsérvese que en este caso el control para decidir si se sale o no del bucle está antes de ***sentencia***, por lo que es posible que **sentencia** no se llegue a ejecutar ni una sola vez.

En el ejemplo propuesto en el apartado 2.4.1 , se estableció el siguiente vector:

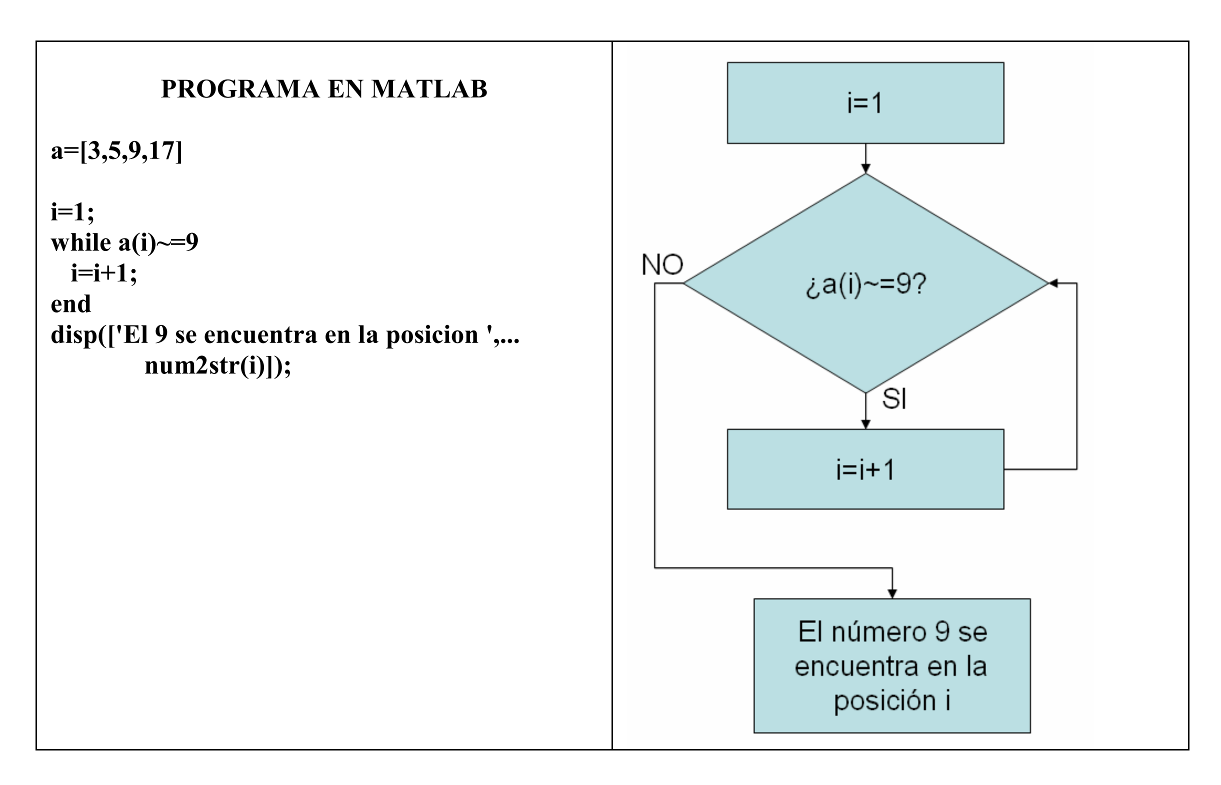
```
>> a=[3,5,9,17]
```

Supongamos ahora que sabemos que dentro de este vector se encuentra el **número 9, pero no sabemos qué posición ocupa.** Si quisiéramos encontrar la posición que ocupa el número **9** dentro del vector **a**, tenemos que mirar, elemento a elemento, los valores del vector, hasta que localicemos el elemento que contiene el número **9**. Este es un caso en el que se utiliza un bucle WHILE, ya que queremos buscar un elemento y **no sabemos cuántas celdas vamos a tener que mirar.**

Para resolverlo, se inicializa un contador a 1 (llamémosle **i**) y se registran todos los elementos de **a** mientras que resulte que el contador no coincida con la posición del número **9** dentro de **a**, esto es**:**

```
WHILE a(i)~=9
```

La sentencia que ha de ejecutarse dentro del bucle es simplemente incrementar en una unidad el valor del contador, para que en el siguiente paso del bucle, se mire la siguiente celda del vector.



PROGRAMA EN MATLAB

```
a=[3,5,9,17]

i=1;
while a(i)~=9
  i=i+1;
end
disp(['El 9 se encuentra en la posicion ',...
          num2str(i)]);
```


3.8.2 Bucle For

For es de los bucles mas versátiles y utilizados. **Los bucles FOR se utilizan cuando queremos que se repita una serie de instrucciones un número determinado de veces.**

Así como los bucles WHILE se repiten mientras que se cumpla una determinada condición, los bucles FOR se repiten mientras que una variable no adquiera un determinado valor. Así que, cada bucle FOR contiene una variable a la que llamaremos **VARIABLE DEL BUCLE FOR**.

Cuando se programa un bucle FOR, se escribe una primera instrucción con la que se dan los **tres parámetros fundamentales del bucle FOR**.

1º **El VALOR INICIAL de la variable del bucle**. Este será el valor que tendrá la variable la primera vez que ejecutan las instrucciones que se encuentran dentro del bucle.

2º **El VALOR FINAL de la variable del bucle.** Esta es la "condición de chequeo particular" de los bucles FOR. Mientras que la **variable del bucle FOR** sea **MENOR O IGUAL** al valor final, se ejecutarán las instrucciones que se encuentran dentro del bucle.

3º **El PASO del bucle.** Esta es una herramienta fundamental dentro del bucle FOR y que hace que sea muy diferente a un bucle WHILE. Y es que la variable del bucle **MODIFICA SU VALOR AUTOMÁTICAMENTE POR CADA PASO DEL BUCLE.** Y ¿cómo lo modifica? Pues lo modifica valiéndose del **PASO,** de forma que, por cada vez que se ejecutan las instrucciones que se encuentran dentro del bucle FOR,

se incrementa **AUTOMÁTICAMENTE** la **variable del bucle FOR** un valor igual al **PASO**.

La expresión general en MATLAB de un bucle FOR es la siguiente:

```
for Variable del bucle=Valor Incial : PASO : Valor Final
   Sentencia;
end
```

Supongamos queremos ejecutar una serie de instrucciones 10 veces. Si llamamos **I** a la variable del bucle FOR**,** la manera más ortodoxa de programar el bucle es la siguiente:

```
for I=1 : 1 : 10
   instrucciones;
end
```

No obstante, se consigue el mismo efecto con las siguientes expresiones:

for I=1 : 10 : 100 **instrucciones;** **end**	**for I=100 : -10 : 1** **instrucciones;** **end**	**for I=10 : -1 : 1** **instrucciones;** **end**

En todos los casos se consigue que la secuencia ***instrucciones*** se repita 10 veces. Sin embargo en muchas ocasiones interesa utilizar la variable **I**. Por ejemplo, cuando se quieren recorrer los elementos de un vector o de una matriz. O cuando se quiere utilizar un número durante una serie de veces y que el valor del número cambie con un determinado paso (como cuando sucede al calcular el factorial de un número). En todos estos casos, se establece un bucle FOR y se utiliza la propia **variable del bucle**.

Un ejemplo típico en el que interesa utilizar la variable del bucle FOR puede ser el producto escalar de dos vectores **a** y **b** de dimensión **n**:

```
pe=0
for i=1:1:n
    pe = pe + a(i)*b(i);
end
```

Primero se inicializa la variable **pe** a cero y la variable i a 1; el ciclo se repetirá mientras que **i** sea menor o igual que **n**, y al final de cada ciclo el valor de **i** se incrementará en una unidad. En total, el bucle se repetirá n veces. La ventaja de la construcción **for** sobre la construcción **while** equivalente está en que en la cabecera de la construcción **for** se tiene toda la información sobre como se inicializan, controlan y actualizan las variables del bucle.

3.8.3 Anidamiento de bucles

Los bucles pueden anidarse unos dentro de otros. Esta estrategia se utiliza muy a menudo. Un ejemplo claro se encuentra a la hora de recorrer los elementos de una matriz. Supongamos que interesa sumar todos los elementos de una matriz de nombre **A.** Para ello vamos a inicializar una variable **S** a cero y, sobre esa variable, iremos sumando todos lo elementos de la matriz **A**. Para acceder a todos los elementos de **A** vamos a utilizar dos variables auxiliares (**F** y **C**) que serán dos números enteros que servirán para referirnos a la fila y columna de **A.** Si **A** es una matriz de 10 filas y 23 columnas, la manera más lógica de sumar todos sus elementos es anidando dos bucles

FOR para controlar la fila y la columna a la que nos referimos, según el siguiente ejemplo:

```
S=0;
for F=1:1:10
  for C=1:1:23
        S=S+A(F,C);
  end
end
```

Capítulo 4 Funciones o procedimientos

4.1 Introducción

Una función es un bloque autónomo e independiente de código que se encarga de realizar una operación concreta. Puede recibir datos de otras partes del programa en forma de parámetros y puede devolver uno o más valores al terminar de ejecutarse.

Pero, ¿para qué emplear funciones?

Pues sobre todo para organizar mejor el programa y para evitar tediosas repeticiones de programación. Si se escribe una sola vez la función apropiada, se puede emplear esta función tantas veces como se desee en diferentes situaciones y localizaciones de un programa, lo que permite modular la programación.

4.2 Funciones o procedimientos

Las funciones (o subrutinas, o subprogramas) son un conjunto de código independiente. Las funciones sirven para aplicar la estrategia "divide y vencerás", ya que mediante las funciones se consigue dividir un programa largo en programas pequeños que son fáciles de realizar. Además, a la hora de sacar los errores de un programa grande es más fácil chequear las funciones separadamente buscando cuál es la función que falla en lugar de revisar un programa de muchísimas líneas de código donde estas interactúan entre sí y donde lo único que se sabe es que algo no funciona como era de esperar.

Para entender y aprender el manejo de funciones veamos un ejemplo: cómo “funciona” la función ***inv()*** de Matlab. Se trata de una función que calcula la matriz inversa de otra matriz, siempre que esta sea regular.

Primero se crea la matriz A, que es la matriz cuya inversa se quiere conocer:

```
» A=[1 4 -3; 2 1 5; -2 5 3]
A =
    1 4 -3
    2 1 5
    -2 5 3
```

A continuación, se calcula la inversa de la matriz A mediante la función ***inv()*** y se asigna a una nueva matriz B:

```
B=inv(A)
B =
    0.1803 0.2213 -0.1885
    0.1311 0.0246 0.0902
    -0.0984 0.1066 0.0574
```

¿Qué ha ocurrido? ***Inv()*** es una función que recibe unos datos, realiza una serie de cálculos, y devuelve los resultados de esos cálculos. En este caso recibe una matriz – la matriz A –, el computador realiza internamente una serie de operaciones y obtiene los elementos que tendría la matriz inversa de A, y luego devuelve esos elementos, que son asignados a otra matriz - la matriz B -.

Si observa detenidamente el proceso, se dará cuenta de que sucede algo así como cuando mete una moneda en una máquina de tabaco. Usted le da una cosa a la máquina (2 €)[3] y ésta le devuelve otra cosa (un paquete de Fortuna con 19 cigarrillos).



Esquema con el intercambio de datos en una función.

En el caso de la función ***inv()*** lo que usted aporta es una matriz y lo que la máquina le devuelve es la inversa de esa matriz. A lo que se aporta se le llama **argumentos de entrada** y lo que devuelve se le llama **valores de retorno** o **argumentos de salida.**

En el caso de la función ***inv()*** el número de **argumentos de entrada** era de uno (la matriz de la que se quiere obtener su inversa), pero hay otras funciones que requieren más de un argumento de entrada, como la función ***power(A,B)*** que lo que hace es elevar A a la potencia B.

Del mismo modo, la función ***inv()*** aporta un único **valor de retorno** (la inversa de la matriz) pero las hay que devuelven más de un valor, como la función ***size(A),*** que tiene dos valores de retorno, como son el número de filas y el número de columnas de la matriz A.

Conocer cuáles son los argumentos de entrada y de salida de una función es **VITAL**, ya que, en realidad, es **lo único importante** que hay que saber de una función. Tanto es así que si escribe por ejemplo: >>**help size**, lo único de lo que se habla es de cuáles son los argumentos de entrada y de salida.

Ejemplo 1. Función matemática

function valor=funcion(x)

El **argumento de entrada** (x) se corresponde con el punto en el que se quiere conocer el valor de la función.

El **argumento de salida** (valor) es el valor de la función en dicho punto.

Internamente, la función se debe corresponder con $f(x) = x^2 - x - \sin(x - 0.15)$. El programa queda como sigue:

```
function valor = funcion(x)
valor = x^2-x-sin(x-0.15);
```

Ejemplo 2. Derivada numérica de la anterior función matemática

Se va a realizar una derivación numérica. Para ello se recurre, siguiendo la misma filosofía que con la integración numérica, a la definición de derivada:

$$Derivada = \frac{f(x+k) - f(x)}{k}$$

Donde *k* es un valor muy pequeño, por ejemplo 1e-6.

[3] Esto era cierto hasta el 16 de Mayo de 2004

En este caso se van a utilizar dos funciones. La primera contendrá una función matemática y la segunda realizará un par de llamadas a la primera función para calcular la derivada de la función en un punto.

function valor=derivada(*x*)

Los **argumentos de entrada** son:

***x*:** punto en el que se quiere calcular la derivada

El **argumento de salida** (valor) es el valor de la derivada de la función en dicho punto. El código correspondiente es:

```
function valor = derivada(x)
k = 1e-6;
derivada = (funcion(x+k)-funcion(x))/k;
```

Ejemplo 3. Función que obtiene las raíces de un polinomio de 2º grado.

A continuación vamos a realizar una función que calcule las raíces de un polinomio de segundo grado. Esta función necesitará 3 argumentos de entrada (los valores a, b y c del polinomio) y 2 de salida (las dos raíces del polinomio). Guarde las siguientes instrucciones en un archivo llamado **funcion_raices_ecuacion.m**, y compruebe que funciona:

```
function [x1,x2]=funcion_raices_ecuacion(a,b,c)
x1=(-b+(b^2-4*a*c)^.5)/(2*a);
x2=(-b-(b^2-4*a*c)^.5)/(2*a);
```

Pruebe la función programada para los siguientes casos:

a=1; b=3; c=2; **Resultado**: 2 raíces reales distintas.

a=1; b=2; c=1; **Resultado**: 2 raíces reales iguales.

a=1; b=1; c=1; **Resultado**: 2 raíces imaginarias.

Capítulo 5 Algoritmos

5.1 Introducción

Parece ser que el origen de la palabra "algoritmo" no está del todo claro. Según la Real Academia de la Lengua, quizá provenga del latín tardío (algobarismus). Otra teoría es que la palabra "algoritmo" tiene su origen en el nombre de un matemático árabe del siglo IX llamado Al-Khuwarizmi, quien estaba interesado en resolver ciertos problemas aritméticos y que concretó una serie de métodos para resolverlos. Estos métodos eran una lista de instrucciones específicas que, de seguirlas obedientemente, se lograba resolver el problema. Quienes abogan por esta teoría defienden que hoy en día, en honor a Al-Khuwarizmi, se denomina algoritmo al conjunto de instrucciones que, de ser seguidas, se logra realizar correctamente una tarea.

Estas instrucciones han de ser entendibles por quien tiene que ejecutarlas, aunque quien tenga que ejecutarlas les vea sentido alguno. **La característica más importante de un proceso algorítmico es que no sea ambiguo**, de manera que cada una de las instrucciones que lo constituyen debe significar solamente una cosa posible. De forma que los algoritmos son una herramienta interesantísima para programar un ordenador. A pesar de lo "inteligentes" que aparentan ser las computadoras, éstas no deciden nada por sí solas, sino que se limitan a ejecutar las instrucciones que se han programado en ellas. Esta manera de funcionar hace que los programadores, antes de escribir las instrucciones para un ordenador, planteen el algoritmo a realizar en un lenguaje "pseudo humano". Asimismo enlazan instrucciones de manera global entre sí mediante diagramas de flujo, concretando el orden de ejecución de las instrucciones.

5.2 Algoritmos directos

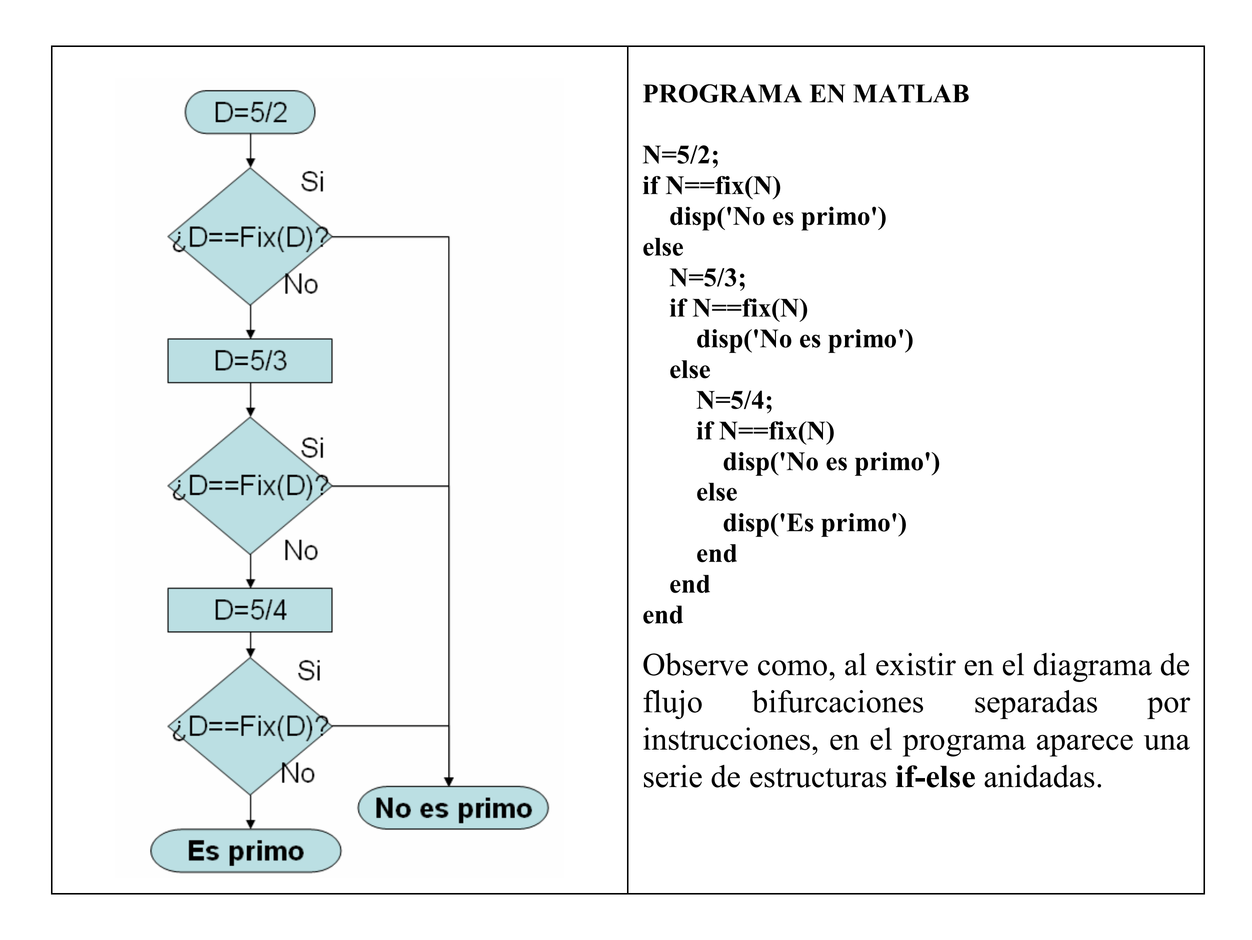
Los primeros algoritmos que van a ser presentados son los algoritmos directos. Por un algoritmo directo se entiende aquel que se compone por un número fijo de pasos.

La característica más importante de los algoritmos directos es que el flujo del programa es **unidireccional** y no se repite ninguno de sus pasos.

Ejemplo 4. Algoritmo para saber si el número 5 es primo

Un algoritmo para determinar si el número 5 es primo puede consistir en dividir al número 5 entre todos los números naturales entre el 2 y el 4. Si alguna división es exacta significará que el número 5 **no es primo**. Para chequear si alguna división es exacta lo que se va a optar es por guardar cada división en una variable llamada D y comparar a D con la parte entera de la propia variable D -para lo que se utilizará la función FIX. Si al realizar alguna división se detecta que D es un número entero (no tiene decimales), puede concluirse que el número 5 **no es primo**, mientras que si después de realizar todas las divisiones resulta que en ningún momento D es un número entero, se concluye que el número 5 **es primo**.

El diagrama de flujo del algoritmo descrito y el programa que lo ejecuta queda de la siguiente manera:



PROGRAMA EN MATLAB

```
N=5/2;
if N==fix(N)
   disp('No es primo')
else
   N=5/3;
   if N==fix(N)
      disp('No es primo')
   else
      N=5/4;
      if N==fix(N)
         disp('No es primo')
      else
         disp('Es primo')
      end
   end
end
```

Observe como, al existir en el diagrama de flujo bifurcaciones separadas por instrucciones, en el programa aparece una serie de estructuras **if-else** anidadas.

Ejemplo 5. Determinar el tipo de raíces de un polinomio de segundo grado

Las raíces de un polinomio de segundo grado (x_1 y x_2) expresado como $P(x) = ax^2 + bx + c$ se obtienen mediante la fórmula siguiente:

$$x_1 = \frac{-b + \sqrt{b^2 - 4ac}}{2a} \qquad x_2 = \frac{-b - \sqrt{b^2 - 4ac}}{2a} \tag{5.1}$$

De forma que se dan tres posibilidades en las que:

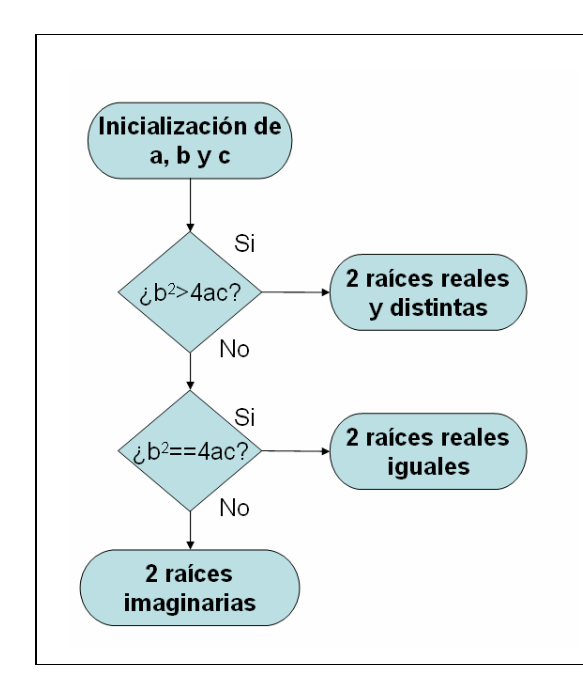
Si el contenido de la raíz es mayor que 0, las raíces son reales y distintas.

Si el contenido de la raíz es igual a cero, las raíces son reales e iguales.

Si el contenido de la raíz es menor que 0, las raíces son imaginarias.

Con lo que el algoritmo para determinar el tipo de raíces de un polinomio de segundo grado puede plantearse de manera directa chequeando cuál de estas tres posibilidades es la que se cumple.

A continuación se presenta el diagrama de flujo y el programa:



PROGRAMA EN MATLAB

```
function raices_polinomio(a,b,c)
if b^2>4*a*c
   disp('2 raices reales y distintas')
elseif b^2==4*a*c
   disp('2 raices reales iguales')
else
   disp('2 raices imaginarias')
end
```

Observe como, al existir en el diagrama de flujo dos bifurcaciones seguidas se establece una estructura **if-elseif-else** en el programa.

5.3 Algoritmos iterativos

Un componente básico de los algoritmos es la iteración. Esta palabra **implica repetición**, ya que iterar significa ejecutar repetidamente algunos pasos elementales de un algoritmo **acercándose, paso a paso, a la solución**. Para realizar un programa de ordenador es muy conveniente localizar los pasos que se repiten y anidarlos dentro de un sistema iterativo.

En el ejemplo anterior en el que se determinaba si el número 5 era o no primo, se ha seguido un método que funciona para el caso concreto del número 5. Si se quisiera desarrollar un programa más genérico (por ejemplo, un programa que determinara si un número **N** es o no primo) haría falta identificar la parte repetitiva del algoritmo y anidarla dentro de un proceso iteración en el algoritmo.

Ejemplo 6. Determinar iterativamente si un número N es o no primo

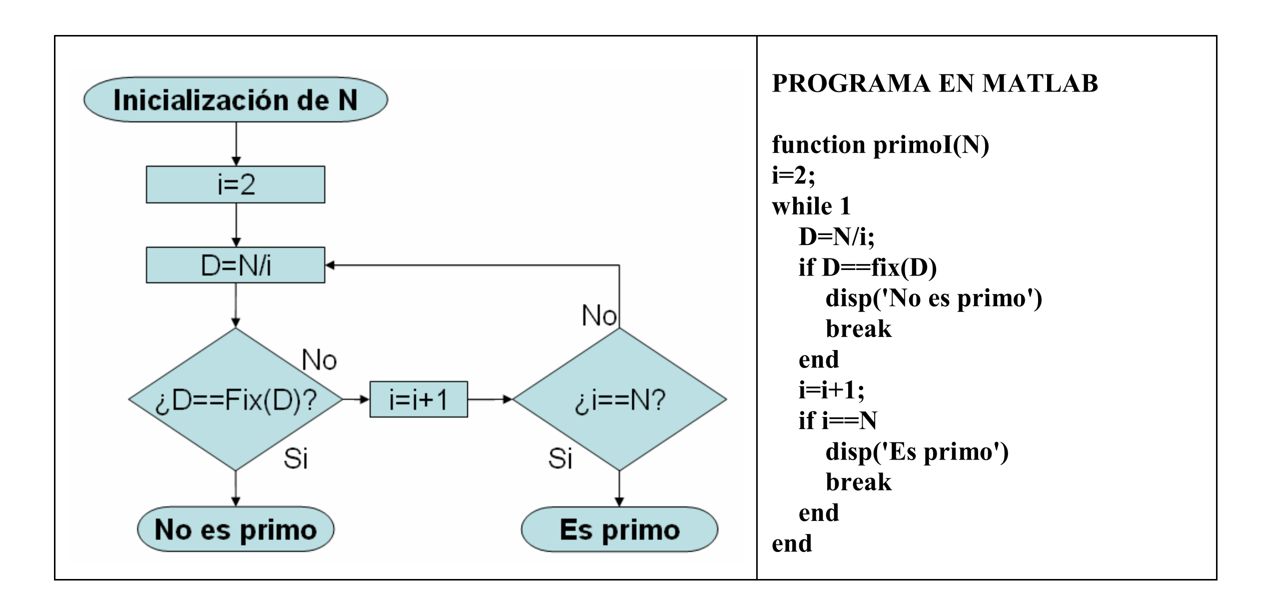
Vamos a rehacer el algoritmo para calcular si el número 5 es primo de manera que sea un algoritmo válido para un número genérico **N**. En el diagrama de flujo del ejemplo para el número 5 se observan tres repeticiones:

Se divide el número **N** entre un número natural.

Se observa si la división es entera.

Se incremento del número natural que va a dividir al número **N**.

En el caso que estamos tratando hemos localizado tres pasos que se repiten, los cuales se realizan desde el número 2 hasta el número 4 en el ejemplo concreto del número 5. De lo que se trata es de anidar estos tres pasos dentro de un bucle que se ejecute mientras que se cumpla una cierta condición; siendo la condición que el número natural que divide a **N** sea inferior a **N**. Vamos a llamar al número que divide **i**, al resultado de la división **D** y al número a chequear **N.** El diagrama de flujo resultante es el siguiente:



PROGRAMA EN MATLAB

```
function primoI(N)
i=2;
while 1
   D=N/i;
   if D==fix(D)
      disp('No es primo')
      break
   end
   i=i+1;
   if i==N
      disp('Es primo')
      break
   end
end
```

En el diagrama de flujo hay tres instrucciones que se repiten iterativamente (la tres que se han descrito anteriormente) y aparece una condición de chequeo para saber si debe detenerse el algoritmo o no (si **i** ha llegado a obtener el valor de **N**). La manera elegida para programar el bucle ha sido la de anidarlo dentro de un bucle WHILE con la condición de chequeo en TRUE (while 1). Este bucle resultaría ser un bucle infinito (ya que la condición de chequeo siempre se cumple) si no fuera porque dentro del bucle se han anidado dos bifurcaciones IF que contienen una ruptura del bucle (BREAK). Esta ruptura se produce cuando la condición de alguna de las dos bifurcaciones se cumple, las cuales son:

1ª Si se encuentra una división exacta (**N** no es primo).

2ª Si resulta que el divisor a probar (i) ha llegado a ser igual a **N** (**N** es primo).

Nota: aunque la estructura seleccionada para establecer el proceso recursivo ha sido un bucle WHILE, quizá sea más propio utilizar un bucle FOR porque se incrementa automáticamente la variable **i** y porque el proceso tiene un final claro, que es cuando **i** llegue a valer **N-1**. El programa con un bucle FOR quedaría así:

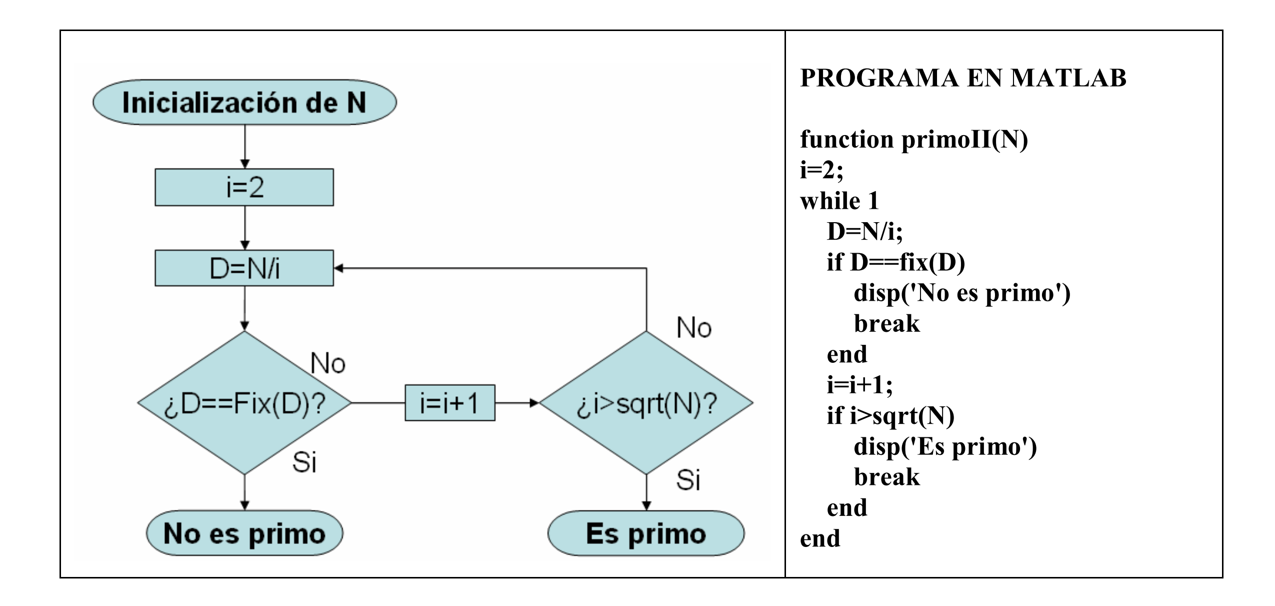
PROGRAMA EN MATLAB

```
function primoI(N)
for i=2:N-1
   D=N/i;
   if D==fix(D)
      disp('No es primo')
      break
   end
   if i==N-1
      disp('Es primo')
   end
end
```

Ejemplo 7. Mejora del algoritmo para calcular si un número N es primo

A continuación se van a presentar dos mejoras del anterior proceso iterativo desde el punto de vista de la computación. Como ya sabemos, para determinar si, por ejemplo, el número 231 es primo, hay que dividirlo entre todos los números naturales entre el 2 y el 230. Pero también es cierto que todos aquellos números naturales superiores al $\sqrt{231}$ no dividen de forma entera al 231, por lo que la condición de chequeo puede

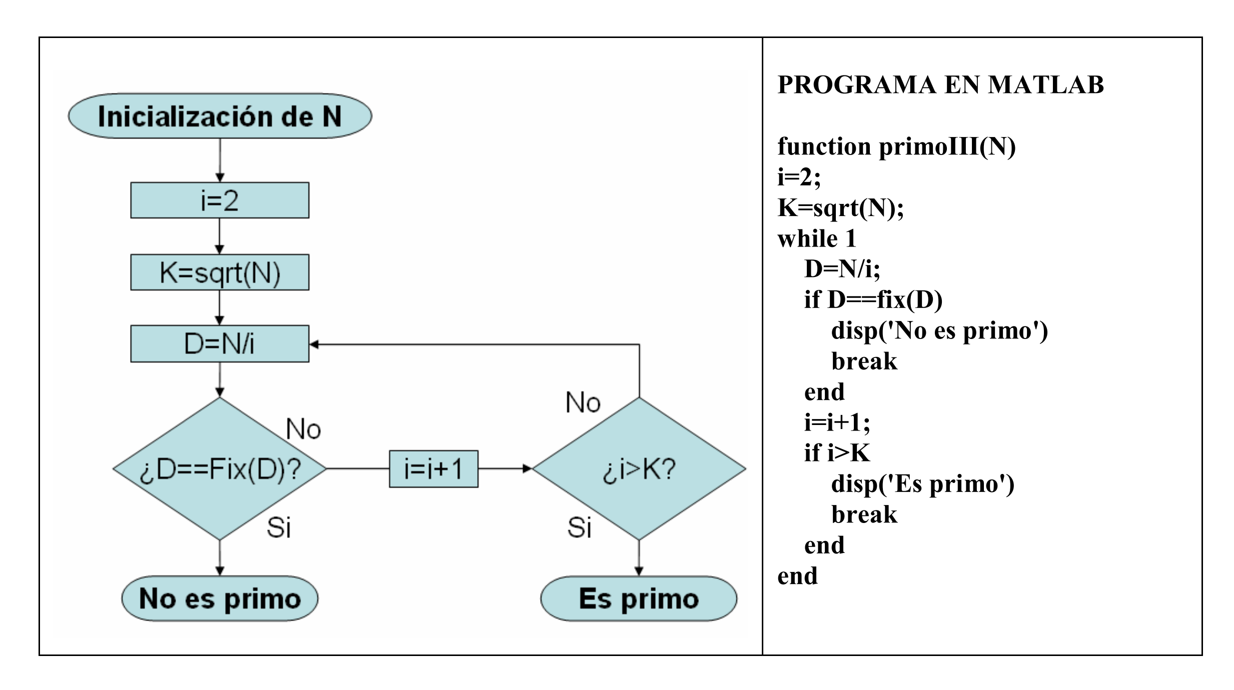
modificarse por comprobar que la variable **i** no rebase el valor de $\sqrt{231}$. Con esto se consigue que, en el caso del número 231, en lugar de repetir el proceso iterativo 229 veces (desde i=2 hasta i=230), se repita solo 14 veces (desde i=2 hasta i=15), ya que por encima del 15 es seguro que no existe ningún número que divida al 231 de manera entera, con lo que se reduce la carga computacional del algoritmo.



PROGRAMA EN MATLAB

```
function primoII(N)
i=2;
while 1
   D=N/i;
   if D==fix(D)
      disp('No es primo')
      break
   end
   i=i+1;
   if i>sqrt(N)
      disp('Es primo')
      break
   end
end
```

Otra mejora sobre el algoritmo anterior consiste en evitar que en cada paso del bucle iterativo se calcule la raíz cuadrada de **N**, ya que calcular la raíz cuadrada de un número es muy costoso para un ordenador. Para ello basta con calcular una vez la raíz de N, almacenarla en una variable (llamémosle **K**) y utilizar esta variable en la condición de chequeo, como puede verse en el diagrama de flujo siguiente.



PROGRAMA EN MATLAB

```
function primoIII(N)
i=2;
K=sqrt(N);
while 1
   D=N/i;
   if D==fix(D)
      disp('No es primo')
      break
   end
   i=i+1;
   if i>K
      disp('Es primo')
      break
   end
end
```

Si se prueba a cronometrar los dos últimos programas presentados (***PrimoII*** y ***PrimoIII***) puede observarse que ***PrimoIII*** es mucho más rápido que ***PrimoII***. ¿Cómo cronometrar un programa? Generalmente se utiliza el propio reloj del ordenador para

medir el tiempo. Además, cuando son programas cuyo tiempo de ejecución es muy corto, se le suele pedir al ordenador que los ejecute varias veces (por ejemplo 1000) y se establece una media del tiempo que ha tardado, ya que medir lo que tarda el programa una sola vez es muy impreciso.

Ejemplo 8. Obtención del factorial de un número N iterativamente

Un ejemplo claro de un proceso iterativo es el cálculo del factorial de un número **N**. La regla del factorial de un número expresada matemáticamente es:

$$\begin{cases} n! = n*(n-1)*(n-2)*...*2*1 \\ 0! = 1 \end{cases} \quad (5.2)$$

De forma que para calcular el factorial del número 5 se siguen los siguientes pasos:

Se multiplica a 2 por 3 y se almacena el resultado.

Se multiplica el resultado anterior por 4 y se almacena de nuevo.

Se multiplica el resultado anterior por 5 y se almacena de nuevo.

Este último resultado es el factorial de 5. Si observamos el proceso se ve necesario utilizar una variable que varíe su valor de uno en uno desde el 2 hasta el 5 (llamémosle a esta variable **i**) Asimismo se necesita otra variable para almacenar el resultado de la multiplicación (llamémosle a esta variable **Resultado**).

Los pasos que se repiten son 3:

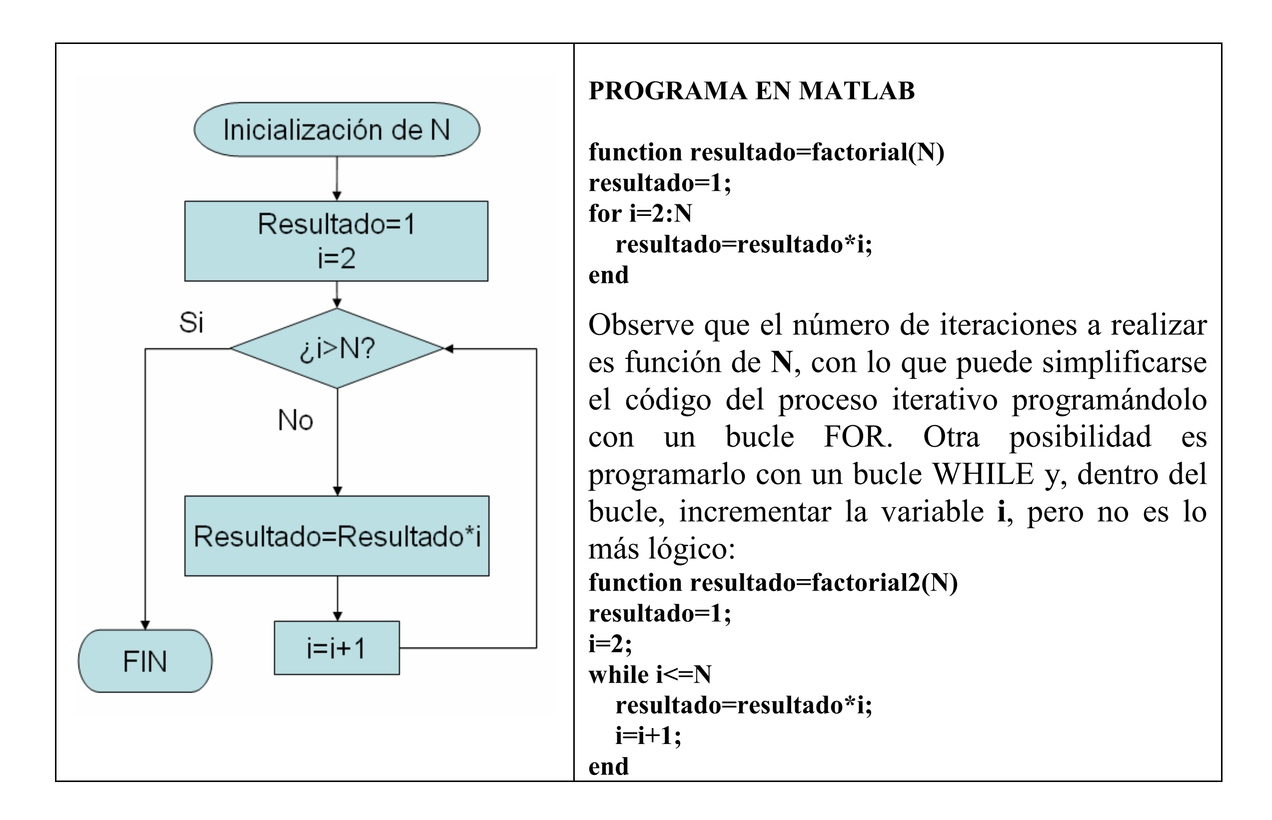
1º Multiplicación de **Resultado** por la variable **i.**

2º Almacenamiento de la operación anterior.

3º Incremento de la variable **i** en una unidad.

En realidad los pasos 1º y 2º se pueden hacer en una sola operación "informática" (multiplicar **Resultado** por **i**) y asignar el resultado de la multiplicación a **Resultado**).

Por último faltaría la condición de chequeo para continuar o para detener los tres pasos durante el proceso iterativo. La condición de chequeo en este caso es comprobar que la variable **i** no supera el valor del número del que se calcula el factorial **N.**



PROGRAMA EN MATLAB

```
function resultado=factorial(N)
resultado=1;
for i=2:N
   resultado=resultado*i;
end
```

Observe que el número de iteraciones a realizar es función de **N**, con lo que puede simplificarse el código del proceso iterativo programándolo con un bucle FOR. Otra posibilidad es programarlo con un bucle WHILE y, dentro del bucle, incrementar la variable **i**, pero no es lo más lógico:

```
function resultado=factorial2(N)
resultado=1;
i=2;
while i<=N
   resultado=resultado*i;
   i=i+1;
end
```

Ejemplo 9. Descomposición factorial de un número N.

Obtener la descomposición factorial de un número natural **N** consiste en determinar todos aquellos números naturales de menor valor que multiplicados entre si dan como resultado el número **N**. Para ello hay que probar a dividir el número **N** entre todos los números naturales que hay entre 2 y $\sqrt{N}$. A continuación se describen los pasos que se repetirán en el proceso iterativo:

Siendo **I** el número con el que se prueba a dividir N en un momento dado, hay que chequear que el valor de I no supere el valor de $\sqrt{N}$:

Si lo supera, se termina el proceso iterativo.

Si no lo supera, se comprueba si **N** es divisible de manera entera entre **I**:

Si **N** no es divisible entre **I** hay que pasar al siguiente número (el **I+1**).

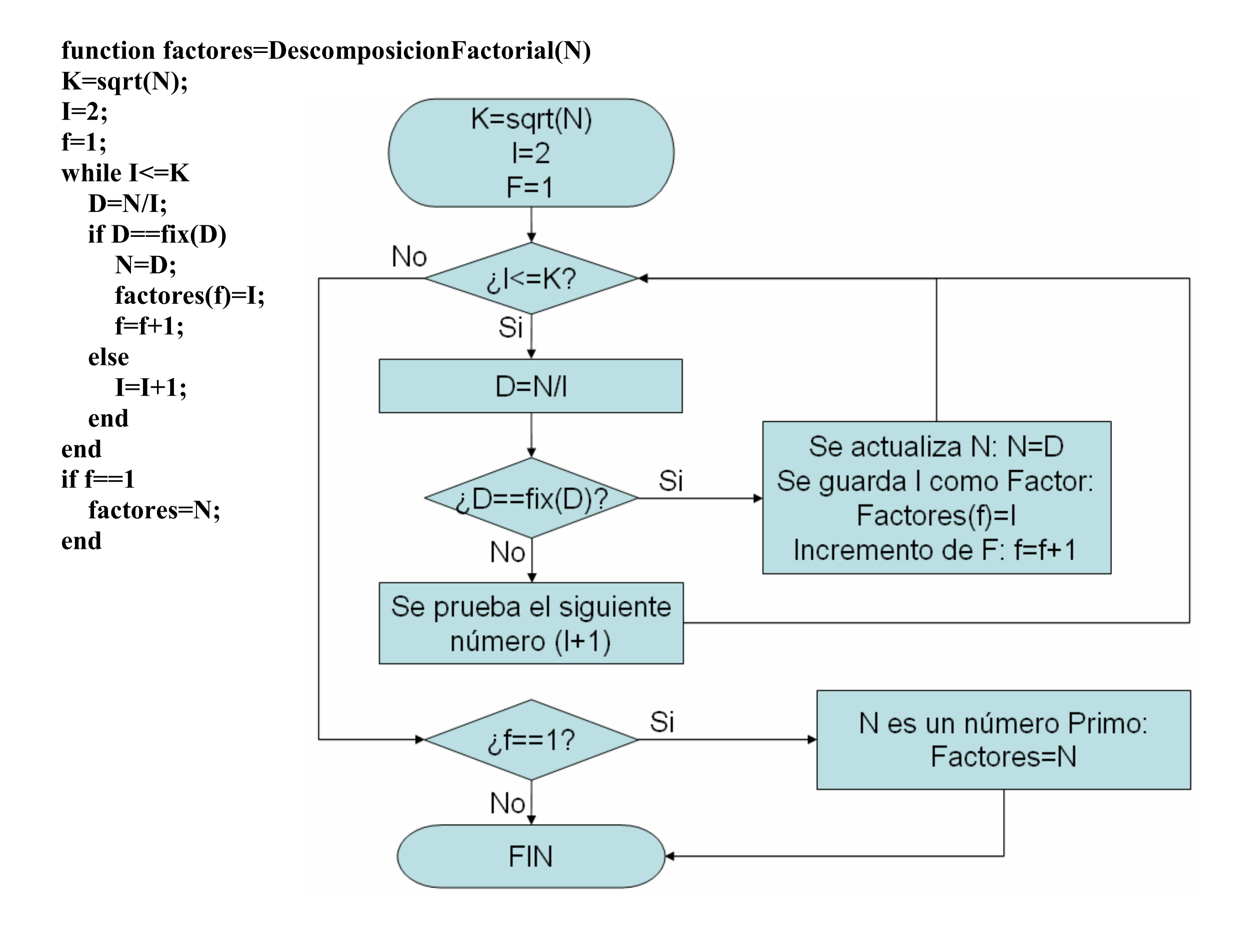
Si **N** es divisible entre **I** hay que:

Almacenar a **I** como factor del número **N**

Actualizar el valor de **N** como **N=N/I**

Los factores se van a guardar en un vector que llamaremos **factores**. Para guardar ordenadamente los factores nos valdremos de una variable auxiliar que llamaremos **f**. Inicialmente esta variable **f** valdrá 1 y, cada vez que aparezca un nuevo factor de **N**, se incrementará en una unidad el valor de **f**. De esta manera, sirviéndonos de la variable auxiliar **f**, podremos guardar el primer factor en la primera celda del vector **factores**, el segundo factor en la segunda celda y así sucesivamente.

```
function factores=DescomposicionFactorial(N)
K=sqrt(N);
I=2;
f=1;
while I<=K
   D=N/I;
   if D==fix(D)
      N=D;
      factores(f)=I;
      f=f+1;
   else
      I=I+1;
   end
end
if f==1
   factores=N;
end
```



5.4 Algoritmos recursivos

Se dice que un objeto es recursivo cuando el propio objeto está compuesto parcialmente por sí mismo. Los sistemas recursivos tienen una importancia particular para realizar ciertas definiciones matemáticas. Un ejemplo lo encontramos en la definición del factorial de un número. La función factorial para los números naturales puede ser expresada de la manera siguiente:

$$\begin{cases} n! = n*(n-1)! \\ 0! = 1 \end{cases} \tag{5.3}$$

Esta es la expresión recursiva del factorial de un número, frente a la iterativa, expresada en la ecuación (5.2)

Los sistemas recursivos no se encuentran sólo en imaginario mundo de las matemáticas, sino que también en la más prosaica realidad, como puede verse al enfrentar dos espejos o al mirar una coliflor o un helecho. Poder establecer una expresión recursiva implica poder definir una serie infinita de objetos mediante la inicialización de un número finito de objetos. Desde el punto de vista informático esto abre posibilidades que dan vértigo, ya que se pueden describir un número infinito de operaciones mediante un programa finito, aunque, al contrario que los programas que contienen bucles, no presente repeticiones de código explícitas. Los algoritmos recursivos, sin embargo, son particularmente adecuados cuando el problema a resolver, o la función matemática a computar o la estructura de datos a ser procesada está definido en términos recursivos. En general un programa recursivo P es función (Π) de una serie de instrucciones S_i distintos de P y de P mismo:

$$P \equiv \Pi\left[S_i, P\right] \tag{5.4}$$

La herramienta necesaria y suficiente para poder expresar programas de manera recursiva en un determinado lenguaje de programación es que el lenguaje de programación admita funciones, procedimientos o subrutinas (como prefiera llamarse). Cuando una función contiene una llamada explícita a sí misma, se dice entonces que hay una recursividad directa. Cuando, por el contrario, una función no realiza una llamada directa a sí misma, sino que lo hace a través de otra función, se dice que tiene una recursividad indirecta.

Ejemplo 10. Obtención del factorial de un número N recursivamente

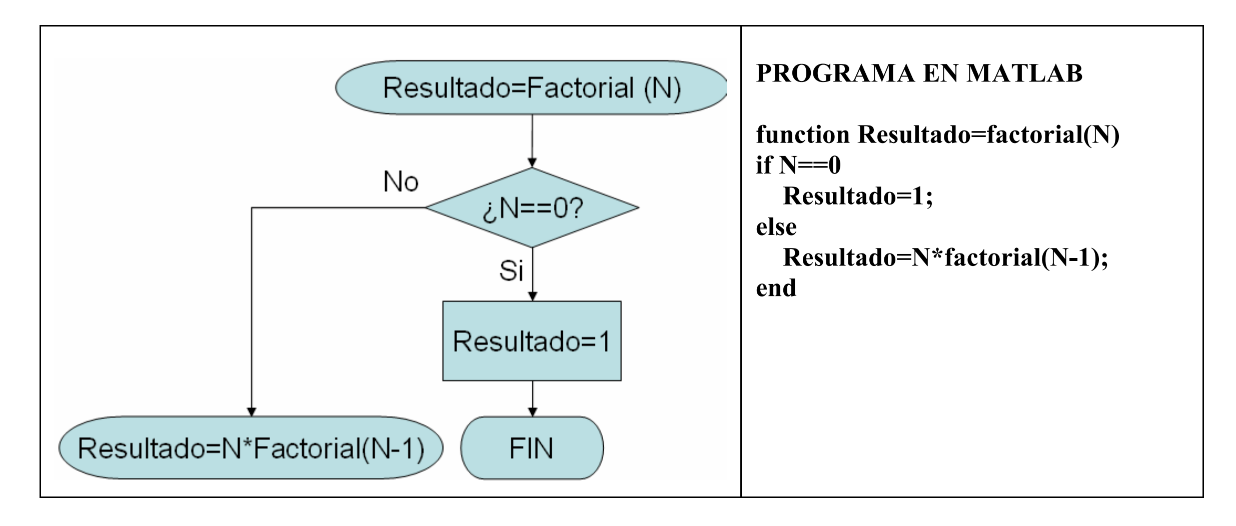
Anteriormente se determinó el factorial de un número N de manera iterativa según la fórmula:

$$\begin{cases} n! = n*(n-1)*(n-2)*...*2*1 \\ 0! = 1 \end{cases}$$

Y acabamos de ver cómo también puede expresarse de manera recursiva según esta otra forma:

$$\begin{cases} n! = n*(n-1)! \\ 0! = 1 \end{cases}$$

La cual define el factorial de un número como el producto de ese número con el factorial del número anterior; salvo el factorial de 0, que es 1 por definición. El diagrama de flujo del algoritmo para calcular el factorial de un número de manera recursiva es el siguiente:



PROGRAMA EN MATLAB

```
function Resultado=factorial(N)
if N==0
   Resultado=1;
else
   Resultado=N*factorial(N-1);
end
```

Hay que observar que en el diagrama de flujo no hay un retroceso en la línea del programa, sino que lo que aparece es una nueva llamada al propio programa (**Factorial**) cada vez que se trata de calcular el factorial de un número superior a 0. A su vez, esta llamada pide calcular el factorial del número anterior al que se encuentra en cada momento, por lo que llegará un momento en el que se tratará de calcular el factorial de 0. En este momento, debido a la bifurcación que hay, se determina el primer resultado, que es 1, y finaliza el programa en el que se está calculando el factorial de 0. Al finalizar este programa se continúa con el cálculo del factorial de 1, donde se establece

el segundo resultado (1*1) y finaliza este segundo programa. A su vez vuelve a continuar el programa superior, que se corresponde con el cálculo del factorial de 2, y así continúa la ascensión sucesivamente hasta llegar a determinar el valor del factorial del número que se ha solicitado.

Ejemplo 11. Determinar recursivamente si un número N es o no primo

Recordamos que un número entero **N** es primo si es mayor o igual a dos y sus únicos factores son el 1 y el propio número. En el Ejemplo 6 y en el Ejemplo 7 se estableció un método iterativo para determinar si un número cumple esta condición. Este método consiste en dividir al número entre todos los números naturales que hay entre el 2 y la raíz cuadrada del número a estudiar, observando si en algún caso la división era exacta.

De lo que ahora se trata es de seguir el mismo razonamiento anterior de manera recursiva. Para ello se va a plantear el diagrama de flujo en dos partes: la inicialización y el cálculo recursivo.

En la inicialización el usuario pasa el número que quiere saber si es primo, el cual queda guardado en la variable **N**. A continuación se guarda la raíz cuadrada de **N** en una variable global -**K**- y por último se inicializa el proceso recursivo en el que se chequea si es divisible **N** entre algún número natural comprendido entre el 2 y **K**. Para ello se utiliza una función auxiliar que llamaremos ***Mirar_si_es_divisible(N,k)*,** la cual se utiliza en la inicialización para k=2. (Nótese que se está diferenciando entre **K** mayúscula y **k** minúscula).

Esta función auxiliar contendrá los pasos que hay que dar para determinar si **N** es divisible entre **k**. En caso de que no lo sea, realizará otra llamada recursiva a si misma para chequear si **N** es divisible entre **k+1**, hasta que suceda que, o bien **k** es superior a **K** -en cuyo caso **N** es primo- o bien **N** sea divisible por **k** -en cuyo caso **N** no es primo.

El diagrama de flujo queda de la siguiente manera:

```
function Primo(N)
global K;
K=sqrt(N);
Mirar_si_es_divisible(N,2)

function Mirar_si_es_divisible(N,k)
global K;
if k>K
   disp([num2str(N) ' es primo'])
elseif N/k==fix(N/k)
       disp([num2str(N) ' no es primo'])
else
       Mirar_si_es_divisible(N,k+1)
end
```

Para no tener que calcular la raíz cuadrada de N, se ha almacenado ésta en la variable K. Además, para no tener que pasarla constantemente por ventanilla cada vez que se llama a la función ***Mirar_si_es_divisible*****,** se ha definido K como variable GLOBAL, mediante la declaración de la variable realizada dentro de las dos funciones que se utilizan.

Ejemplo 12. El triángulo de Sierpinski

El triángulo de Sierpinski entra en el ámbito de los fractales. Fractal procede del adjetivo latino 'fractus', que significa "interrumpido" o "irregular". La geometría del fractal fue utilizada por Benoît Mandelbrot en 1975 para describir ciertas formas complejas de la naturaleza como nubes, sistemas montañosos, galaxias, flores, ramificaciones arbóreas y bronquiales, rocas, cuencas hidrográficas, el sistema neuronal, las líneas costeras, esponjas,... y objetos matemáticos como el conjunto de Cantor o el triángulo de Sierpinski, cuyo comportamiento caía fuera del marco de la matemática tradicional.

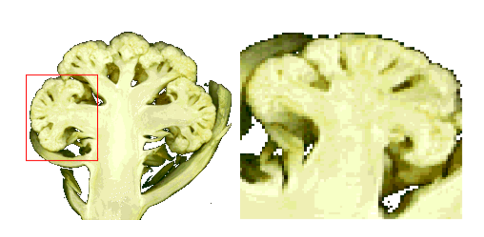


Figura 5.1 Ejemplo de una estructura fractal. Se corresponde con una coliflor cortada transversalmente.

El triángulo de Sierpinski es una construcción geométrica realizada mediante una sucesión de triángulos. La base del método está en que un triángulo puede dividirse en tres triángulos exactamente iguales. A su vez, cada uno de estos tres triángulos puede dividirse en otros tres triángulos exactamente iguales y así sucesivamente. De manera que el triángulo de Sierpinski es un triángulo formado a partir de una sucesión sucesiva de sucesivos triángulos que se suceden sucesivamente, formándose a partir de triángulos pequeños otros triángulos cada vez más grandes.

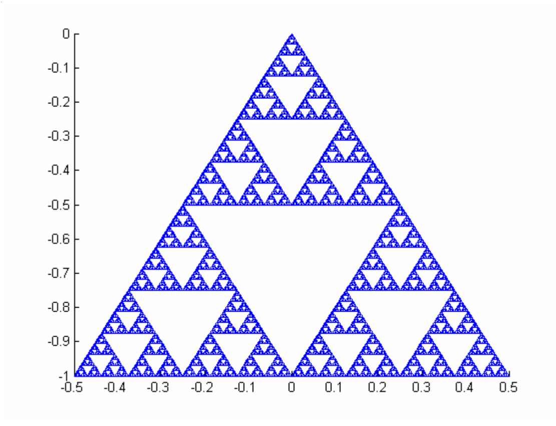


Figura 5.2 Triángulo de Sierpinski de orden 8

La manera adecuada para realizar esta construcción geométrica es mediante un sistema de dibujo recursivo. En la parte inicial del proceso se indica el orden del triángulo de Sierpinski, donde el orden es el número de subdivisiones triangulares con las que se divide el triángulo mayor. Posteriormente se procede a dibujar los triángulos según el orden indicado.

Para dibujar los triángulos se va a establecer una subrutina propia con la que se dibujará un triángulo mediante tres variables **(X,Y,h)**. La variables **X, Y** se corresponden con la posición del vértice superior del triángulo y h se corresponde con la altura o la base del triángulo, como puede verse en la Figura 5.3. Obsérvese que en la Figura 5.3, las proporciones no se mantienen, ya que aunque está dibujado como un triángulo equilátero, es un isósceles. La subrutina para dibujarlo recibirá el nombre de ***Dibujar_triangulo(x,y,h).***

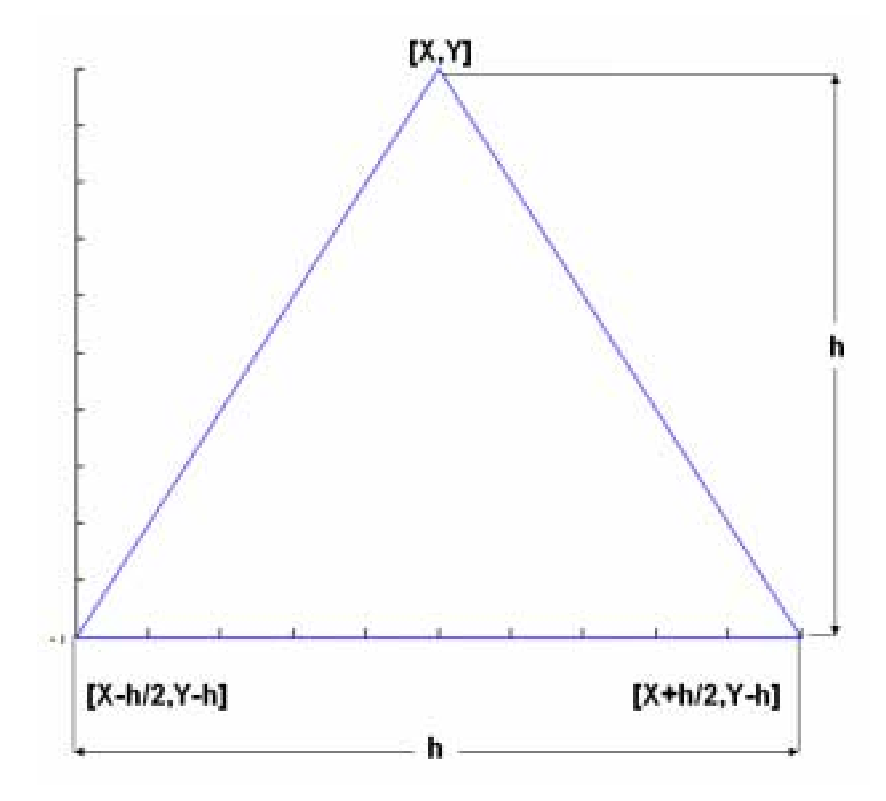


Figura 5.3 Dimensiones para realizar un triángulo

La inicialización la realizaremos dentro de una rutina principal a la que llamaremos ***Triangulo_Sierpinski(N)***, donde **N** es el orden del triángulo. En esta rutina se inicializarán las coordenadas **X, Y** del primero de los triángulos que va a ser dibujado. Por real decreto vamos a hacer que el punto de origen esté en la posición (0,0) -punto A_0 de la Figura 5.4-. Siguiendo el mismo criterio, se asigna una longitud inicial a **h** de una unidad. Por último esta rutina iniciará una subrutina a la que llamaremos ***triangulo(N,x,y,h).*** Esta subrutina es quien contendrá las instrucciones para ir dibujando los triángulos recursivamente.

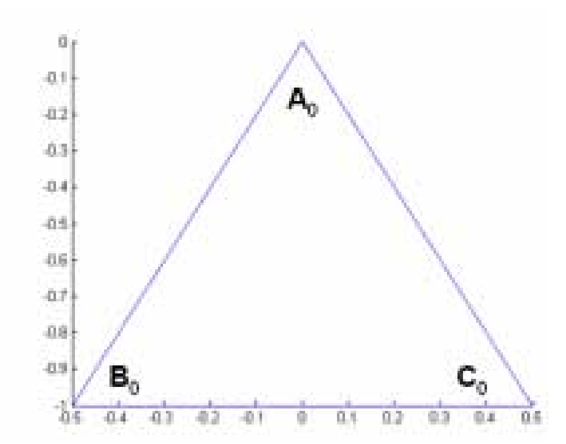


Figura 5.4 Desde cada un de los tres puntos A, B y C se dibujan sucesivamente el resto de triángulos

Básicamente la subrutina ***triangulo(N,X,Y,h)*** se va a limitar a localizar el lugar en el que ha de ser dibujado cada uno de los triángulos, así como de determinar el tamaño que tendrán los triángulos. Como se ha dicho en la introducción al ejemplo, cada triángulo se compone de otros tres triángulos más pequeños. En concreto, un triángulo de altura **h** se compone de otros tres triángulos de altura **h/2,** como puede verse en la Figura 5.5. De forma que el triángulo que tiene su vértice superior en la posición A_i tiene una altura h y está formado por tres triángulos de altura h/2 que tienen su vértice superior en las posiciones A_{i+1}, B_{i+1} y C_{i+1}. Con este esquema queda planteado el sistema recursivo ya que cada vez que se ejecuta la subrutina ***triangulo(N,X,Y,h),*** esta se llama a si misma 3 veces - ***triangulo(N-1,X,Y,h)***- para elaborar un triángulo de orden menor, modificando las posiciones X, Y según el lugar donde debe encontrarse el vértice superior de cada triángulo y reduciendo h a la mitad.

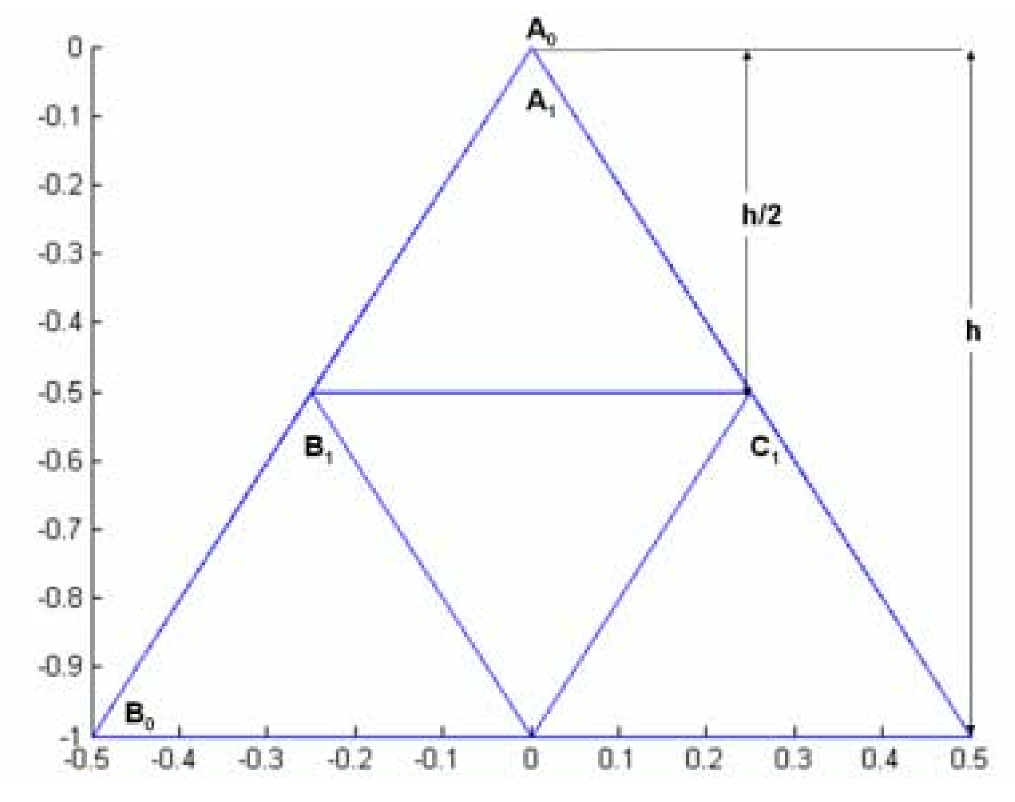


Figura 5.5 Evolución del posicionamiento sucesivo de los tres puntos A, B y C en atención al orden del Triángulo de Sierpinski.

Cuando el orden del triángulo a dibujar es 0, detienen las llamadas recursivas. En ese momento se procede a dibujar el triangulo y finaliza el hilo de esa subrutina. La evolución del dibujo de un triángulo de Sierpinski de orden 3 según el esquema planteado es la siguiente:

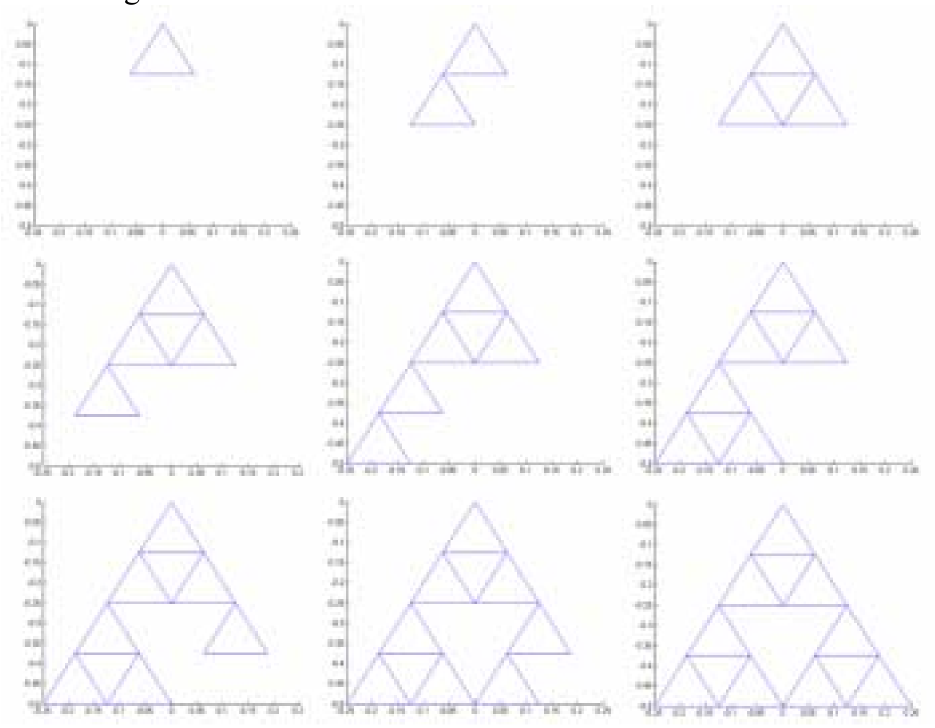
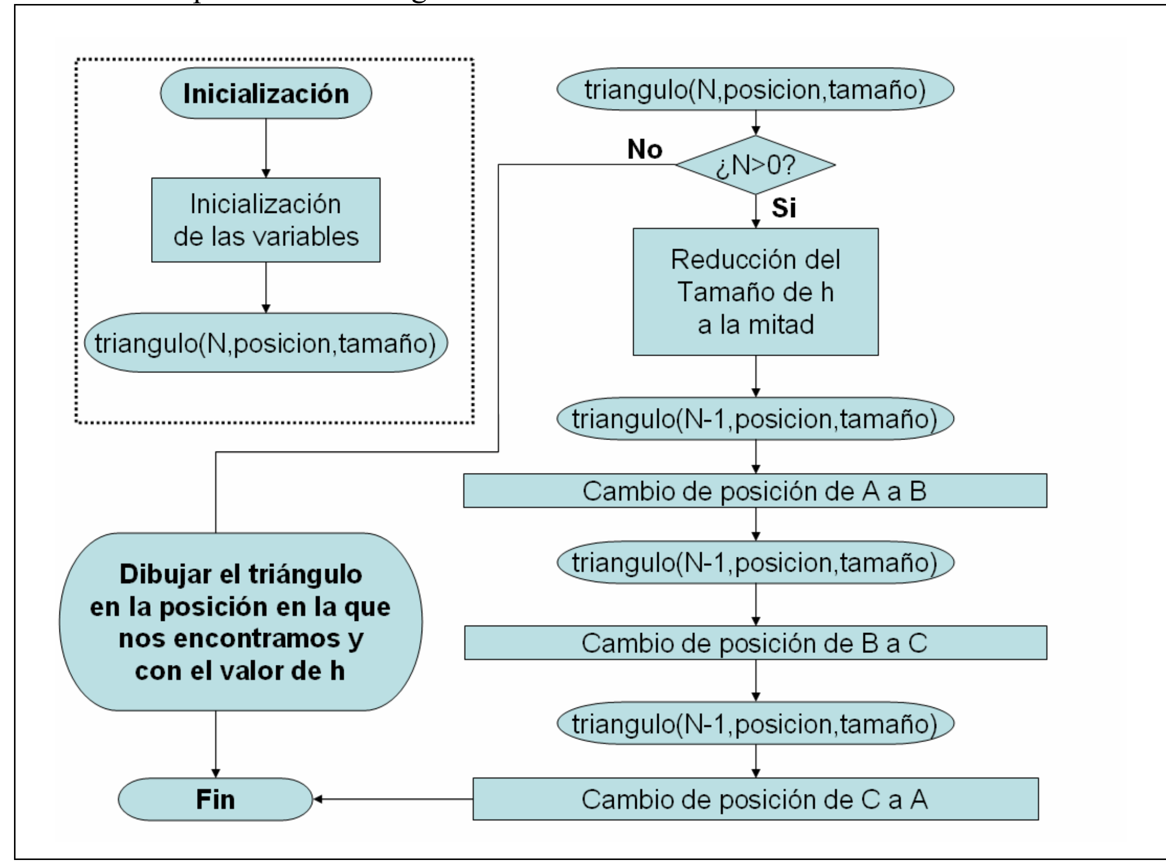


Figura 5.6 Evolución del dibujo de un triángulo de Sierpinski de orden 3

Asimismo, el diagrama de flujo para dibujar el triángulo de Sierpinski de la manera que acaba de ser planteada es el siguiente:



```
function Triangulo_Sierpinski(N)
x=0;
y=0;
h=1;
triangulo(N,x,y,h)

function triangulo(N,x,y,h)
if N>0
   h=h/2;
   triangulo(N-1,x,y,h)
   x=x-h/2;
   y=y-h;
   triangulo(N-1,x,y,h)
   x=x+h;
   triangulo(N-1,x,y,h)
   x=x-h/2;
   y=y+h;
else
   Dibujar_triangulo(x,y,h)
end

function Dibujar_triangulo(x,y,h)
line([x x-h/2],[y y-h])
line([x-h/2 x+h/2],[y-h y-h])
line([x+h/2 x],[y-h y])
```

5.5 Algoritmos de prueba y error

Un área de la programación particularmente intrigante es la referente a la resolución general de problemas. Esta tarea consiste en dar con algoritmos para encontrar soluciones a problemas específicos sin seguir una regla fija para la computación, sino mediante "prueba y error". La manera más común para ello es descomponer el proceso de "prueba y error" en subtareas. A menudo, estas subtareas se expresan de manera natural en forma recursiva probando un número finito de veces las propias subtareas hasta dar con la solución al problema. En este caso se estaría planteando un **esquema recursivo de prueba y error.**

Otras veces, la manera de establecer el esquema para encontrar la solución no se plantea de manera recursiva, sino que se establece una condición que, mientras ésta se cumpla, el programa sigue actuando hasta dar con la solución mediante la repetición sistemática de una serie de pasos. En este caso se estaría planteando un **esquema iterativo de prueba y error.**

5.5.1 Algoritmos iterativos de prueba y error

Ejemplo 13. Método de Newton Rapshon de manera iterativa

El método de Newton-Raphson o simplemente el método de Newton, es uno de los métodos numéricos más poderosos para resolver el problema de búsqueda de raíces de una función matemática expresada en la forma f(x)=0. La base del método reside en el concepto de derivada de una función.

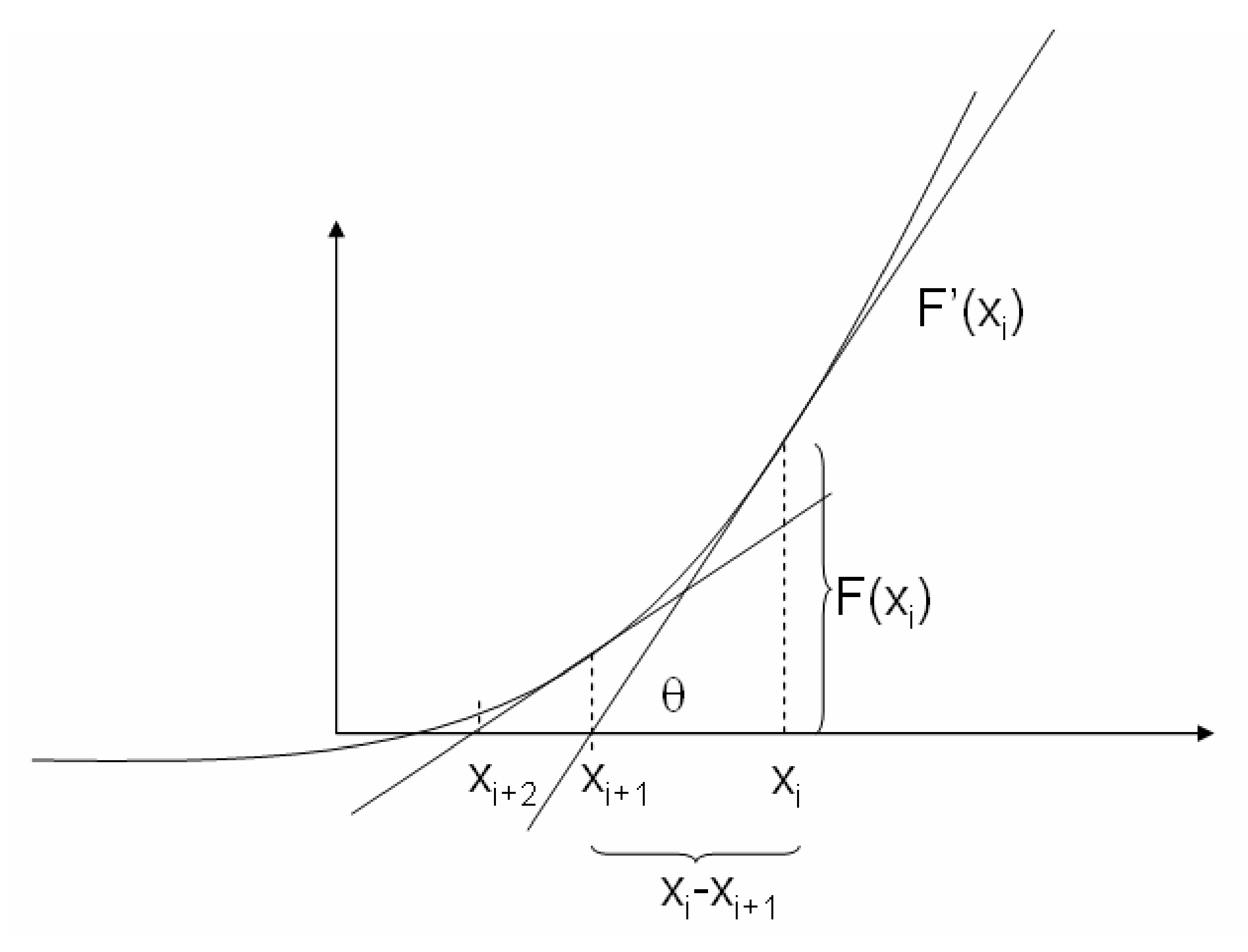


Figura 5.7 Método de Newton-Raphson para obtener las raíces de una función f(x) mediante aproximaciones sucesivas

Esta figura muestra como se obtienen las raíces de una función f(x) mediante tangentes sucesivas. Comenzando con una aproximación inicial x_0, la aproximación x_1 es la intersección con el eje x de la línea tangente a la gráfica de f(x) en $(x_0,f(x_0))$. La aproximación x_2 es la intersección con el eje x de la línea tangente a la gráfica de f(x) en $(x_1,f(x_1))$ y así sucesivamente.

Observa que:

$\tan\theta = f'(x)$=pendiente de la recta que pasa por $(x_i,f(x_i))$.

$\tan\theta$ = Cateto opuesto / Cateto contiguo.

Con esto presente, Newton y Raphson hicieron el siguiente razonamiento matemático:

$$\frac{f(x_i)}{x_i - x_{i-1}} = f'(x_i) \qquad (5.5)$$

$$\frac{x_i - x_{i-1}}{f(x_i)} = \frac{1}{f'(x_i)} \qquad (5.6)$$

$$-x_{i-1} = \frac{f(x_i)}{f'(x_i)} - x_i \qquad (5.7)$$

$$x_{i-1} = x_i - \frac{f(x_i)}{f'(x_i)} \tag{5.8}$$

De manera que para un valor $\mathbf{x_i}$ se determina un nuevo valor $\mathbf{x_{i+1}}$ más próximo a **f(x)=0.**

De lo que se trata ahora es de hacer una función que ejecute iterativamente la operación señalada en la ecuación (5.8). Con esto nunca llegaremos a la solución f(x)=0, pero nos iremos aproximando sucesivamente, de manera que, después de realizar un número suficiente de iteraciones, habremos conseguido un valor de x muy próximo a la raíz. Generalmente no se programa un número concreto de iteraciones, sino que se establece un criterio para decidir cuando debe detenerse el proceso iterativo. Como el objetivo es buscar un valor de x en el que f(x)=0, lo que se establece es que el proceso iterativo continúe mientras que f(x) supere un valor próximo a cero.

Para el ejemplo concreto realizaremos un programa para determinar las raíces de la siguiente función:

$$f(x) = (x^2 - x)\sin(x + 0.15) \tag{5.9}$$

Para ello realizaremos una función de Matlab cuyo encabezado sea:

```
function raiz=Newton_Raphson(xo,error)
```

Donde los **argumentos de entrada** son:

xo: punto de inicio del método.

error: se utilizará para determinar la condición de parada del algoritmo. Es la diferencia entre el valor de f(xi) y 0 y cuando se cumpla que |f(xi)|<|error| se para el método iterativo.

Los **argumentos de salida** son:

raiz: es el valor que tiene xi en el momento en el que se decide parar el algoritmo.

La función ***Newton_Raphson*** deberá realizar la iteración:

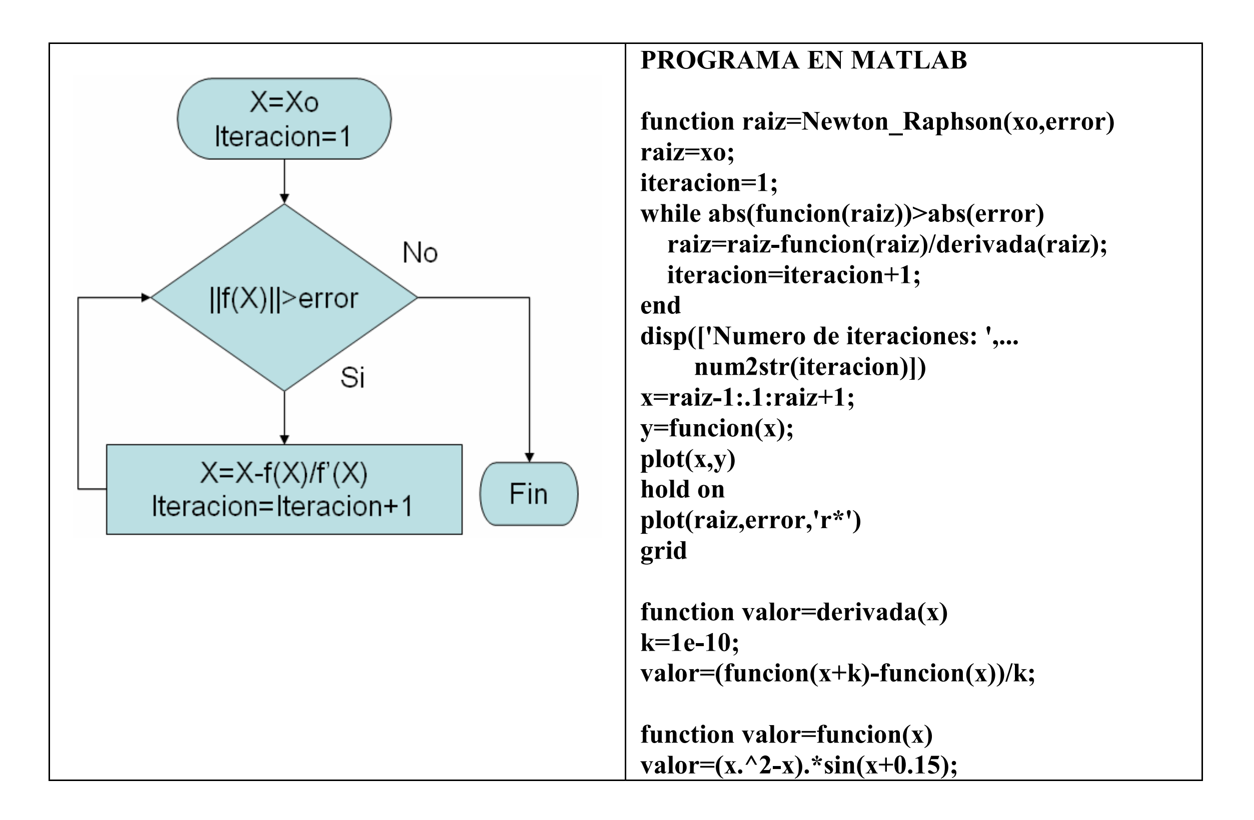
$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)}$$

mientras que se cumpla que:

$$|f(x_i)| > error$$

Para calcular la derivada utilizaremos una constante de derivación de 1e-10. Además le agregaremos al programa un contador que indique cuántas iteraciones ha efectuado antes de pararse. En concreto dibujará unos 20 puntos de la función a lo largo del intervalor **[raiz-1,raiz+1]**, siendo **raiz** el valor de retorno de la función. Incluiremos la función ***grid*** para dibujar un mayado a la gráfica.

El diagrama de flujo y el programa son los siguientes:



PROGRAMA EN MATLAB

```
function raiz=Newton_Raphson(xo,error)
raiz=xo;
iteracion=1;
while abs(funcion(raiz))>abs(error)
   raiz=raiz-funcion(raiz)/derivada(raiz);
   iteracion=iteracion+1;
end
disp(['Numero de iteraciones: ',...
      num2str(iteracion)])
x=raiz-1:.1:raiz+1;
y=funcion(x);
plot(x,y)
hold on
plot(raiz,error,'r*')
grid

function valor=derivada(x)
k=1e-10;
valor=(funcion(x+k)-funcion(x))/k;

function valor=funcion(x)
valor=(x.^2-x).*sin(x+0.15);
```

Ejemplo 14. Cálculo de las dimensiones óptimas de un cilindro.

Supongamos el caso de un jeque árabe que pide que se realice un programa para determinar las dimensiones que debe tener un barril de petróleo para que, albergando un determinado volumen, presente el mínimo área. El problema es que el jeque en cuestión quiere exportar el petróleo en bidones de distinto volumen, por lo que necesita un programa que sea capaz de dimensionar el barril en función del volumen solicitado.

Para solucionar esto vamos a realizar un algoritmo de prueba y error en el que estableceremos un proceso iterativo con el que nos acercaremos sucesivamente a la solución óptima. El algoritmo seguirá los siguientes tres pasos:

1º Se determina la altura **h** que debe tener un **barril inicial** que tenga un radio muy pequeño (por ejemplo **r=0.001**) para albergar el volumen **Vc** solicitado. Esta altura se encontrará lejos de la altura que de unas proporciones óptimas y sabemos que habrá que reducirla para dar con la geometría del barril óptimo.

2º Calcularemos el **área** que presenta este **barril inicial**.

3º Esta área inicial será muy grande. De lo que se tratará ahora es de implementar un sistema iterativo en el que se vaya **incrementando el radio** con un **paso pequeño** (por ejemplo **paso=0.01)** hasta dar con las dimensiones del barril (r y h) que presentan el mínimo área. El criterio para detener las iteraciones es que se lleguemos a aquella iteración en la que demos con un área lateral superior al área lateral que se había obtenido en el paso anterior.

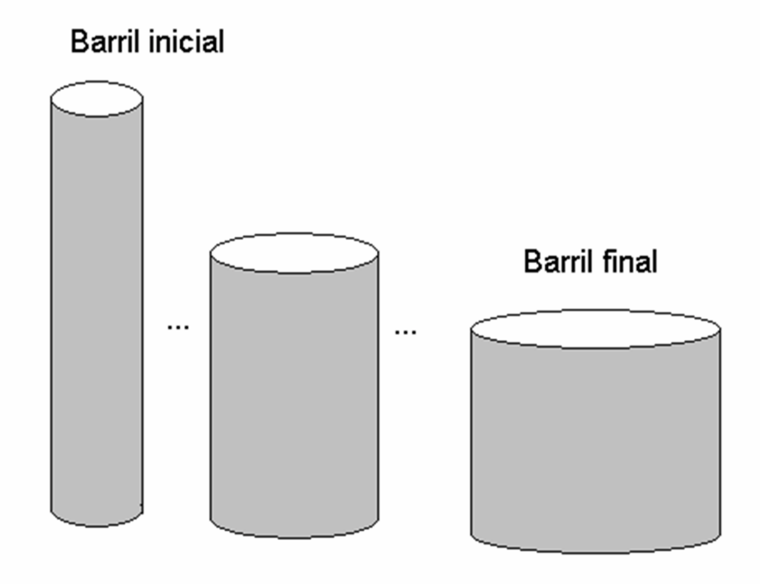


Figura 5.8 Evolución de los distintos barriles. Todos los barriles tienen el mismo volumen, pero distinta área. El barril situado a la izquierda de la figura se corresponde con un barril que tiene mucha altura y mucha área. El proceso de reducción de la altura se mantiene mientras que el área se vaya reduciendo. Cuando el área comience a incrementarse, el proceso se detiene.

Esto es, después del **2º paso** se dispondrá de un primer área (llamémosle **Area**). Ya dentro del proceso iterativo, al incrementar el radio, se podrá calcular un nuevo área para ese radio (llamémosle **Area_nueva)** que podrá compararse con el anterior área. Si resulta que se ha reducido el área, el proceso iterativo debe continuar. Si, por el contrario, el área nueva es peor, debe detenerse el proceso.

Vamos a programarlo en una función de Matlab cuyo encabezado sea:

```
function [r,h]=Area_optima(Vc)
```

Donde **Vc** es el **volumen** que debe albergar el barril y **r** y **h** son el **radio** y la **altura** -calculados por la función- que minimizan el **área** del barril de volumen **Vc.**

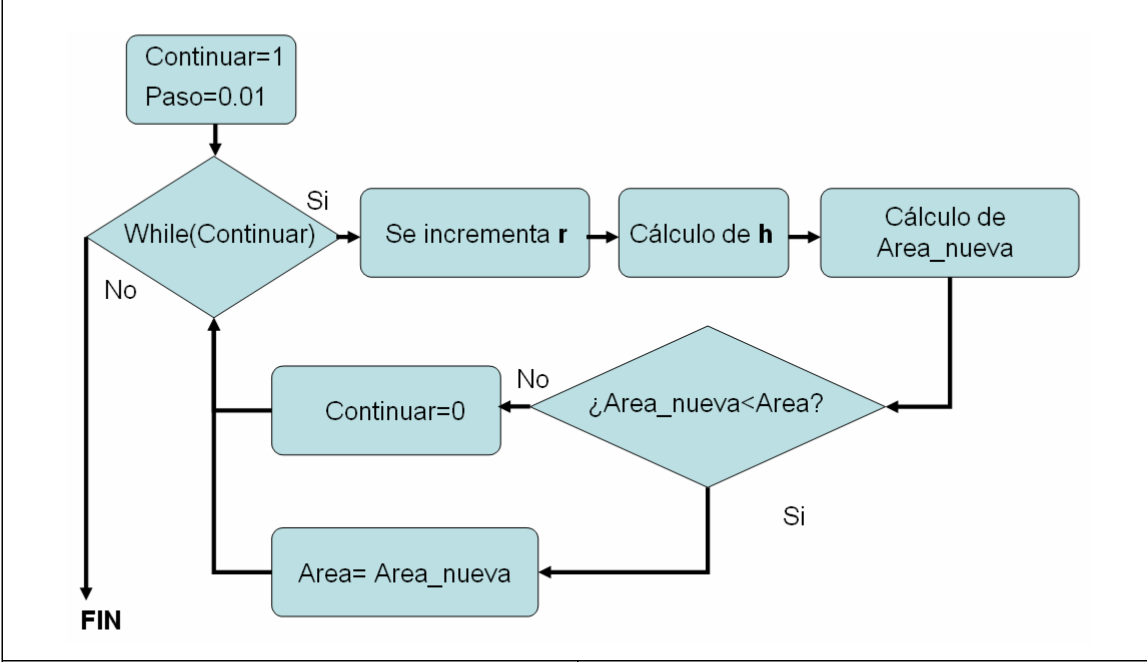
Las fórmulas que hay que considerar son:

Área de la circunferencia: πr^2

Área de un cilindro: $2\pi r^2 + 2\pi rh$

Volumen de un cilindro: $\pi r^2 h$

El diagrama de flujo para el **3er paso** es el siguiente:



PROGRAMA EN MATLAB

```
function [r,h]=Area_optima(Vc)
r=0.1;
Ab=pi*r^2;
h=Vc/Ab;
p=2*pi*r;
A=2*Ab+p*h;
continuar=1;
while(continuar)
   r=r+.01;
   Ab=pi*r^2;
   h=Vc/Ab;
   p=2*pi*r;
   Anueva=2*Ab+p*h;
   if Anueva<A
      A=Anueva;
   else
      continuar=0;
   end
end
```

En este caso, el proceso iterativo ha sido programado mediante un bucle WHILE en el que se chequea el valor de una variable BOOLEANA (**continuar**). El valor de esta variable es 1 inicialmente. Cuando se detecta que el área recién calculada es superior a la obtenida en la anterior iteración, el valor de **continuar** pasa a ser 0, con lo que en el siguiente paso del bucle la condición de chequeo es FALSE y el programa se detiene.

Aún así, la manera más ortodoxa de realizar esta operación es la presentada en el Ejemplo 6 y en el Ejemplo 7, donde se utiliza la sentencia BREAK.

5.5.2 Algoritmos recursivos de prueba y error

Ejemplo 15. Método de Newton-Rapshon de manera recursiva

A continuación se va a rehacer el método de Newton-Raphson de manera recursiva. La manera en la que se plantea es haciendo llamadas sucesivas a la subrutina en la que se realizan las aproximaciones sucesivas pasando el nuevo valor de X que se quiere chequear.

Este valor irá estando cada vez más cerca de la solución f(x)=0 por lo que se añade una bifurcación al comiendo de la subrutina para chequear si se está suficientemente cerca de la solución. Esta bifurcación establece si debe realizarse una nueva llamada a la propia subrutina o bien si debe finalizar el programa.

Newton-Rapshon(xo,error)

||f(xo)||>error

No

Si

Newton-Rapshon(xo-f(xo)/f'(xo),error)

Fin

```
function Newton_Raphson(xo,error)
if abs(funcion(xo))>abs(error)
   Newton_Raphson(xo-funcion(xo)/derivada(xo),error)
else
   disp(['La solucion es ' num2str(xo)])
end
```

Resulta imposible describir todos los algoritmos recursivos de prueba y error. De hecho, tampoco es la intención de este libro, sino que es más bien la de familiarizarse con la computación. Sin embargo un par de algoritmos van a ser descritos para desarrollar el conocimiento en esta disciplina. Un tipo de algoritmos dentro de los algoritmos recursivos de prueba y error son los **algoritmos recursivos de búsqueda "hacia delante y hacia atrás".** A continuación veremos el ejemplo del "viaje del caballo" y el "problema de las 8 reinas". Son dos ejemplos situados en el entorno del juego del ajedrez.



Este juego ha sido objeto de estudio para buscar maneras de programar siguiendo el razonamiento que sigue la inteligencia humana ya que, la manera de pensar que se sigue en este juego es abordable siguiendo métodos mecanicistas.

Ejemplo 16. El viaje del caballo

En ciertos casos se puede plantear un esquema recursivo de prueba y error donde las pruebas que realiza el ordenador construyen una serie caminos de búsqueda de soluciones. Estos caminos de búsqueda dan como resultado un árbol de subtareas, donde algunos caminos no llegan a la solución mientras que otros sí. En muchos problemas, este árbol crece muy rápidamente, normalmente de manera exponencial, con la subsiguiente carga computacional.



Vamos a introducirnos en la técnica descrita mediante el conocido ejemplo del "viaje del caballo". Dado un tablero de ajedrez de NxN casillas, un caballo situado en una determinada casilla debe recorrer todo el tablero -de acuerdo a los movimientos permitidos según el juego del ajedrez- visitando todas las casillas sin situarse dos veces en ninguna de ellas. De forma que el problema consiste en encontrar un camino, si es que existe alguno, que recorra todo el tablero siguiendo la ruta mas corta, y por lo tanto realizando N^2-1 movimientos, siguiendo los movimientos del caballo de ajedrez.

Considerando que el caballo puede realizar 8 movimientos posibles, la manera lógica para resolver este problema es la de construir el camino mediante una sucesión de búsquedas parciales, siendo ésta la búsqueda parcial de una nueva casilla que puede ser visitada. En el caso de encontrarse una nueva casilla, se procede a buscar la siguiente (**se avanza hacia delante**) y, en caso de no encontrase, se vuelve a la casilla de la que se había partido (**se avanza hacia atrás**), para probar una nueva casilla que pueda ser visitada distinta a la que se acababa de probar.

De manera que el procedimiento consistirá en inicializar las variables necesarias y proceder a dar el primer paso. La resolución del problema la realizaremos para un tablero con un número cualquiera de casillas y comenzando en una fila **(f)** y columna **(c)** cualquiera.

Siendo **Casillas**, el número de casillas del tablero, las variables que se inicializan son:

Tablero: es una matriz llena de ceros con un número de celdas igual a CasillasxCasillas. Según vaya el caballo pasando por cada celda se llenará la celda en cuestión con el número que indica en qué paso ha accedido el caballo a ella.

a: es un vector de ocho elementos con los desplazamientos permitidos en las filas del tablero para un caballo de ajedrez.

b: es un vector de ocho elementos con los desplazamientos permitidos en las columnas del tablero para un caballo de ajedrez. Los vectores **a** y **b** están coordinados entre sí, de manera que **a(1)** y **b(1)** representan el desplazamiento en la fila y la columna para el primero de los movimientos y así sucesivamente.

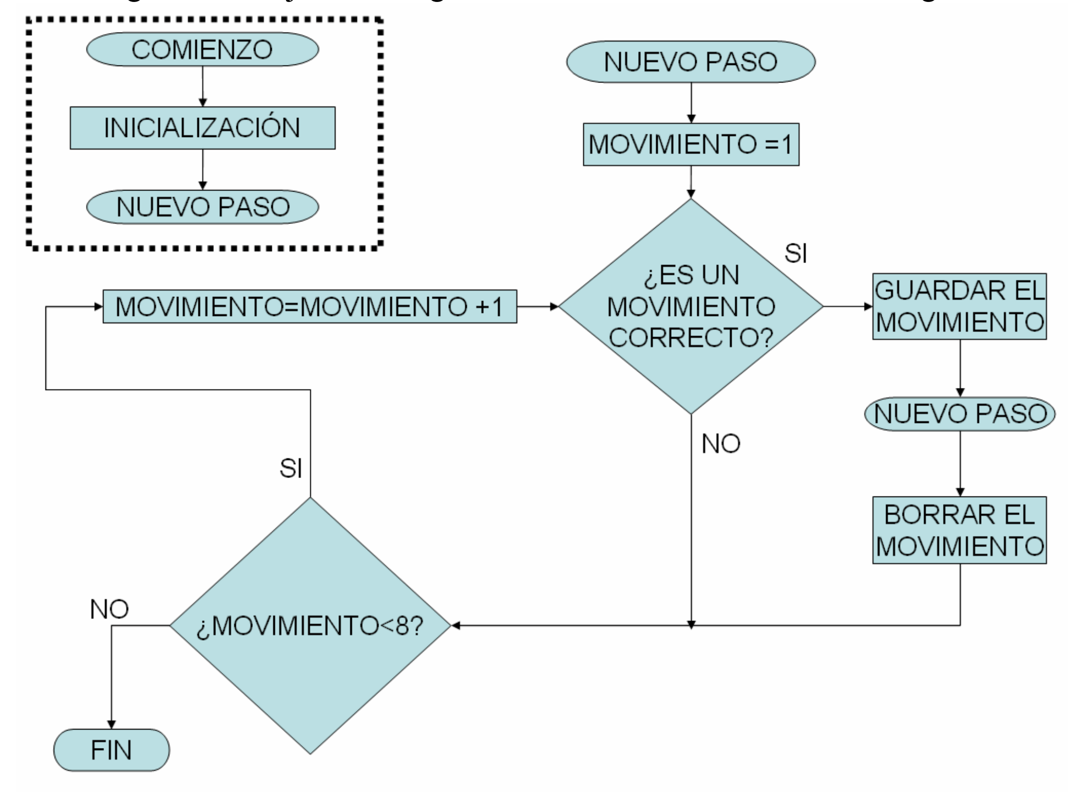
N: se corresponde con el número de movimientos realizados por el caballo. Será una variable auxiliar para determinar si con el movimiento que se acaba de realizar se ha llegado a alguna solución, ya que cuando **N** adquiera el valor de **CasillasxCasillas** significará que el caballo a pasado por todas las casillas.

La inicialización se realizará en la función principal, a la que llamaremos ***El_viaje_del_Caballo(f,c,Casillas).*** Las variables que se pasan por ventanilla son la fila de la que parte el caballo, **f**, la columna, **c,** y el número de filas o columnas que tiene el tablero, **Casillas.** Después de la inicialización se procede a dar el primer paso. Para ello se utilizará la función ***nuevo_paso(Tablero,f,c,a,b,N).***

La función ***nuevo_paso*** será una función recursiva en la que se irá probando cada uno de los 8 movimientos que puede dar el caballo. Para determinar cuál de los ocho movimientos se está probando utilizaremos una variable auxiliar a la que llamaremos **movimiento**.

En primer lugar, la función ***nuevo_paso*** probará a dar el primero de los 8 movimientos. **Si es correcto lo guarda y vuelve a llamarse a sí misma volviendo a repetir el proceso**, con lo que se da un paso adelante**. Si el movimiento probado no es correcto** (esto es, se sale del tablero o accede a una casilla por donde ya ha pasado) **pasa a probar el siguiente movimiento.** Cuando ha probado los 8 movimientos, la función termina y con ella termina también ese camino de búsqueda, con lo que se da un paso hacia atrás.

El diagrama de flujo con el algoritmo descrito anteriormente es el siguiente:



```
function backtraking(f,c,Casillas)
global a
global b
Tablero=zeros(Casillas);
a(1)=2;b(1)=1;
a(2)=1;b(2)=2;
a(3)=-1;b(3)=2;
a(4)=-2;b(4)=1;
a(5)=-2;b(5)=-1;
a(6)=-1;b(6)=-2;
a(7)=1;b(7)=-2;
a(8)=2;b(8)=-1;
N=1;
Tablero(f,c)=N;
i=1;
nuevo_paso(Tablero,f,c,N);

function nuevo_paso(Tablero,f,c,N)
global a
global b
N=N+1;
for i=1:8
   f1=f+a(i);
   c1=c+b(i);
   if 1<=f1 & size(Tablero,1)>=f1 & 1<=c1 & size(Tablero,2)>=c1 & Tablero(f1,c1)==0
      Tablero(f1,c1)=N;
      nuevo_paso(Tablero,f1,c1,N);
      Tablero(f1,c1)=0;
      if N==size(Tablero,1)^2
         Tablero(f1,c1)=N
      end
   end
end
```

La variable **N** es una variable que contabiliza el número de movimientos realizados. No aparece en el diagrama de flujo, pero si en el código del programa. Gracias a esta variable, se soluciona la casuística que aparece cuando se ha cumplido el objetivo de recorrer todas las casillas del tablero y, en consecuencia, no es posible realizar ningún movimiento más. Se puede identificar que el programa ha llegado a este momento porque el valor de **N** coincide con ***size(Tablero,1)^2***. Ahora lo que se hace es sacar por pantalla el tablero con los pasos realizados (***Tablero(f1,c1)=N*** -observe que no se ha puesto '**;**' al final de la instrucción).

Es interesante el recurso empleado para poder utilizar los vectores con los movimientos **a** y **b**. Al ser dos vectores con los valores constantes, se ha definido como *variables globales* tanto en la función de inicialización (**backtraking**) como en la de búsqueda (***nuevo_paso***). Con esto se evita tener que pasar los valores de **a** y **b** por ventanilla en cada una de las llamadas, con lo que la velocidad del proceso mejora.

A continuación se presenta la primera de las múltiples soluciones que se obtienen cuando se le pide al programa que determine el camino a seguir para un tablero de 8x8 situando al caballo inicialmente en la casilla 1,1.

1	60	39	34	31	18	9	64
38	35	32	61	10	63	30	17
59	2	37	40	33	28	19	8
36	49	42	27	62	11	16	29
43	58	3	50	41	24	7	20
48	51	46	55	26	21	12	15
57	44	53	4	23	14	25	6
52	47	56	45	54	5	22	13

Puede observarse (con un poco de paciencia) como los movimientos realizados tienen una correspondencia con el orden en el que se han definido los 8 movimientos en los vecotres **a** y **b**.

Ejemplo 17. El problema de los emparejamientos estables

Imaginemos dos grupos de personas A y B con un mismo número de componentes N. Se trata de encontrar el conjunto de N parejas $\langle a, b \rangle$ siendo a del grupo A y b del grupo B, que satisfagan ciertas restricciones. Se pueden utilizar muchos criterios para establecer estos emparejamientos. Uno de ellos es conocido como la "regla de los emparejamientos estables", el cual se explica a continuación:

Supongamos que **A** es un conjunto de motoristas y **B** un conjunto de cámaras de televisión. El objetivo es tratar de emparejar a los motoristas con los cámaras de televisión para cubrir la retrasmisión de una carrera ciclista. Cada motorista y cada cámara tienen distintas preferencias a la hora de elegir una pareja del otro grupo. Si las **N** parejas se establecen de forma que existe un motorista y un cámara de televisión que se prefieren mutuamente frente a la pareja que les ha tocado respectivamente, entonces **el emparejamiento resulta inestable**. Si, por el contrario, este tipo de sinergias no existen, se dice entonces que **el emparejamiento es estable**. Esta situación caracteriza muchos problemas que se presentan cuando hay que realizar emparejamientos de acuerdo a ciertas preferencias.

Supongamos que acaban de nombrar a un nuevo jefe para cubrir la actualidad deportiva en ese canal de televisión. Este jefe quiere que todas las personas que están a su cargo estén lo más contestas posible. Va a intentar que todas las órdenes que se den estén consensuadas por todos los afectados. Se va a cubrir la retransmisión de una maratón con 8 parejas de motoristas y de cámaras. Para establecer los emparejamientos se ha decidido que cada motorista pase el orden de preferencia que tiene sobre los cámaras y que cada cámara pase el orden de preferencia que tiene sobre los motoristas. Con ello se ha elaborado la siguiente lista con la que hay que establecer los emparejamientos:

		Ranking	1	2	3	4	5	6	7	8
El motorista	1	selecciona al cámara	7	2	6	5	1	3	8	4
	2		4	3	2	6	8	1	7	5
	3		3	2	4	1	8	5	7	6
	4		3	8	4	2	5	6	7	1
	5		8	3	4	5	6	1	7	2
	6		8	7	5	2	4	3	1	6
	7		2	4	6	3	1	7	5	8
	8		6	1	4	2	7	5	3	8
El cámara	1	selecciona al motorista	4	6	2	5	8	1	3	7
	2		8	5	3	1	6	7	4	2
	3		6	8	1	2	3	4	7	5
	4		3	2	4	7	6	8	5	1
	5		6	3	1	4	5	7	2	8
	6		2	1	3	8	7	4	6	5
	7		3	5	7	2	4	1	8	6
	8		7	2	8	4	5	6	3	1

Tabla 5.1 Datos recogidos a los motoristas y a los cámaras de televisión

De forma que el motorista 1 prefiere estar en primer lugar con el cámara 7, en segundo lugar con el cámara 2 y así sucesivamente.

Un camino para dar con la solución es probar todos los pares de miembros de los dos grupos, uno detrás de otro, localizando las soluciones. Se establece, en este caso, una búsqueda de prueba y error hacia delante y hacia atrás sobre las distintas pruebas que se van haciendo. Este es el método seguido en el caso del **problema de las 8 reinas**, presentado en el libro "**Aprenda a programar...**". Llamando **Prueba** a la función para buscar la pareja al motorista **M** según el orden establecido mediante su ranking **R,** podemos escribir el siguiente diagrama de flujo que es completamente coincidente al del problema de las 8 reinas, salvando las pequeñas diferencias debidas a la casuística:

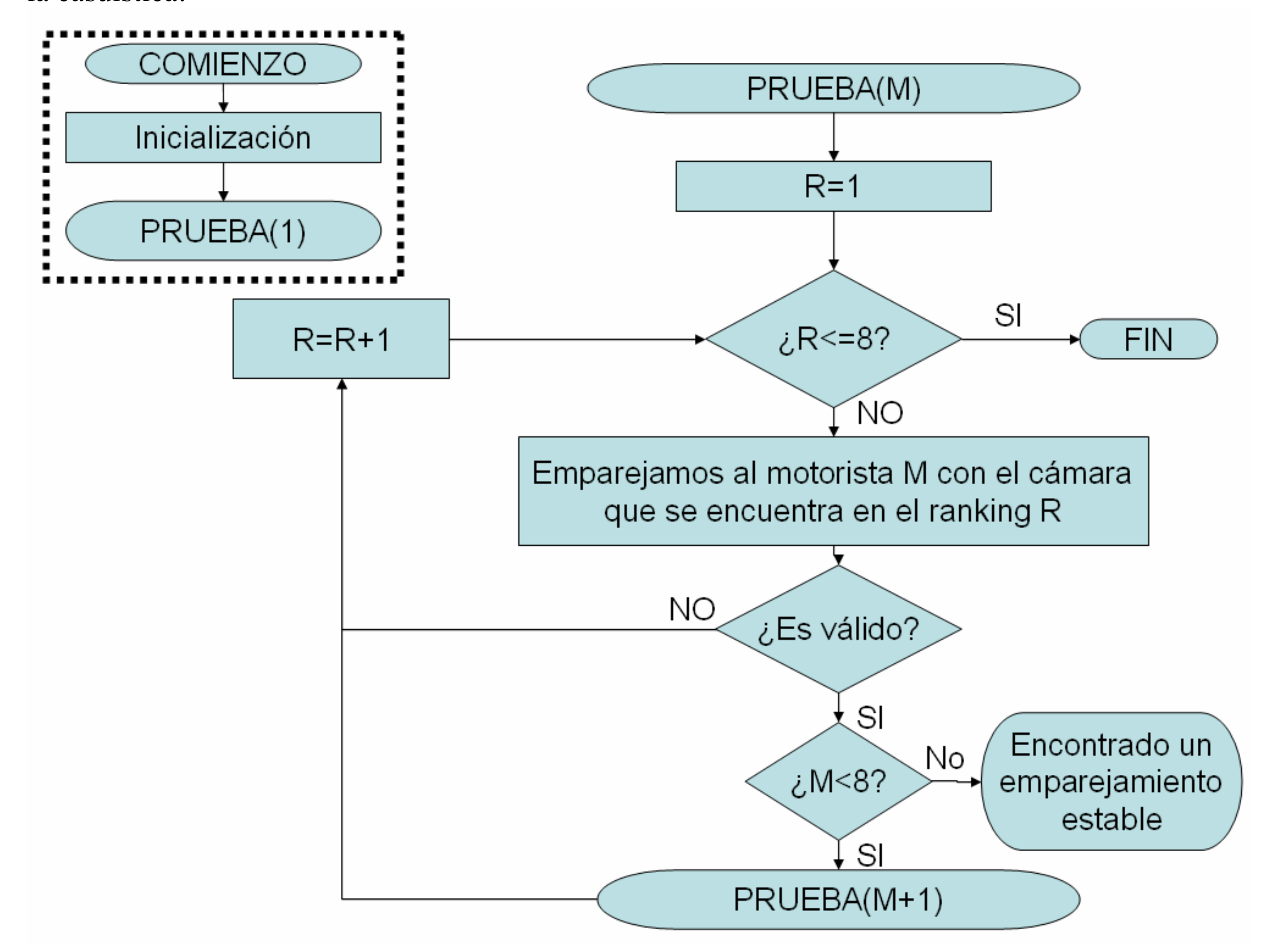


Figura 5.9 Diagrama de flujo del proceso de prueba y error de búsqueda hacia delante y hacia atrás para dar con emparejamientos estables.

En el diagrama de flujo propuesto, se busca a un cámara que pueda ser pareja de un motorista. De manera que comenzamos buscando pareja al motorista 1, luego al motorista 2 y así sucesivamente; sin embargo observe que podría realizarse al revés, buscando primero pareja al cámara 1, luego al 2 etc.

Dentro de la Figura 5.9, la pregunta **¿Es válido?** tiene dos partes. Para que la respuesta sea **sí**, hay que comprobar en primer lugar que el cámara no haya sido ya asignado. Y en segundo lugar hay que comprobar que el emparejamiento es estable.

Para ello hay que decidir cómo agrupar la información. En primer lugar vamos a guardar los datos iniciales que aparecen en la Tabla 5.1 en dos matrices:

MR: Matriz de 8x8 con el ranking establecido por los motoristas.

CR: Matriz de 8x8 con el ranking establecido por los cámaras.

Por ejemplo, **MR(3,2)** se corresponde con el cámara preferido en segundo lugar por el motorista 3, que es el cámara 2, según la Tabla 5.1.

Vamos a guardar el resultado de la búsqueda en un vector llamado **X** de 8 elementos en el que se indiquen ordenadamente el cámara elegido por el motorista de cada casilla. Así por ejemplo, X(3) se correspondería con el cámara asignado al motorista 3.

Además, va a ser conveniente establecer un segundo vector **Y** de otros 8 elementos donde guardaremos ordenadamente a los motoristas asignados a los distintos cámaras. Lógicamente, no es necesario crear **Y,** ya que es en realidad una información redundante. En realidad se puede crear **Y** a partir de **X**, y viceversa, según la relación:

$$X\left(Y\left(C\right)\right)=C$$

$$Y\left(X\left(M\right)\right)=M$$

Sin embargo, el disponer de **Y** va a mejorar sensiblemente la eficiencia del algoritmo.

Con esto, el diagrama de flujo de la Figura 5.9 puede ser reescrito de la siguiente manera:

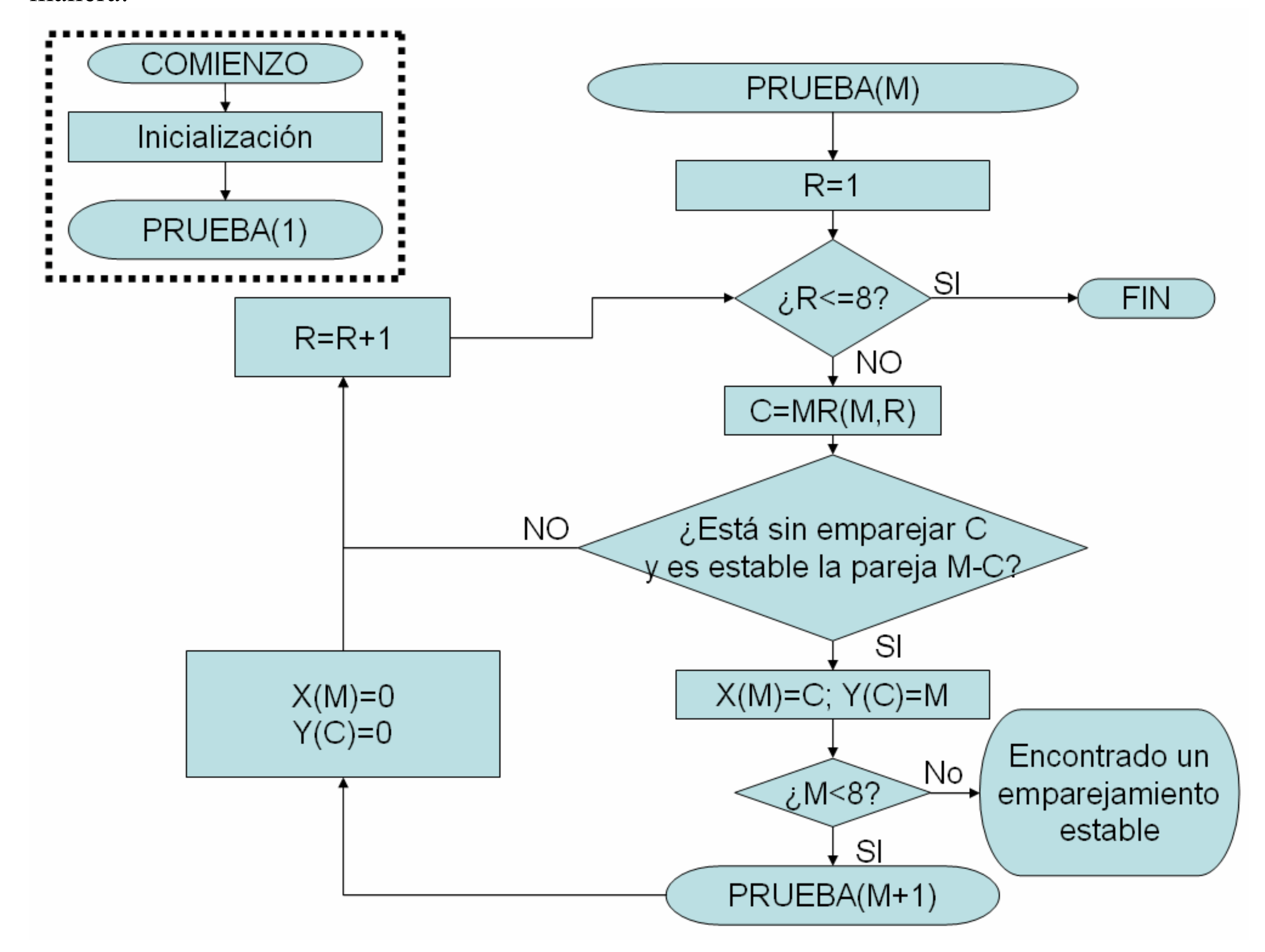


Figura 5.10 Adaptación del diagrama de la Figura 5.9 concretado para las variables propuestas.

Una parte fundamental en el diagrama de la Figura 5.10 es determinar si es estable la nueva pareja M-C establecida. Desgraciadamente no es posible establecer si es estable la pareja con una simple comprobación como en el caso del problema de las 8 reinas. Lo primero que tenemos que tener en cuenta es que la estabilidad o no de una pareja se determina según el ranking establecido en las matrices **MR** y **CR**. Sin embargo, éste ranking no está expresado de manera explícita en estas dos matrices. Lo interesante sería disponer de matrices en las que sí estuviera explicito de manera que se pudiera acceder directamente al ranking que tiene el motorista M sobre el cámara C y viceversa.

Para ello vamos a establecer dos matrices auxiliares que son función directa de **MR** y **CR**. Estas matrices son **RMC** y **RCM.** La primera matriz (**RMC)** contiene el ranking establecido por los motoristas sobre los cámaras. De manera que **RMC(3,1)** es el ranking que ocupa el cámara 1 según el orden de preferencia del motorista 3 (que es 4, según el ejemplo propuesto en la Tabla 5.1). Así mismo la segunda matriz, **RCM**, contiene el ranking establecido por los cámaras sobre los motoristas. Con estas dos matrices auxiliares se facilita muchísimo la determinación de la estabilidad o no de una nueva pareja.

El proceso para saber si es o no estable un emparejamiento se construye directamente aplicando la definición de la estabilidad. Un emparejamiento <M,C> no es estable por dos posibilidades:

1º Existe un cámara, distinto a C, que prefiere a M, frente al motorista que le han asignado y, a su vez, M le prefiere a él frente a C.

2º Existe un motorista distinto a M, que prefiere a C, frente al cámara que le han asignado y, a su vez, C le prefiere a él frente a M.

Lo que ahora toca es realizar una función cuyo encabezado sea:

function [XM,YM]=EmparejamientoEstable(MR,CR)

Donde **MR** y **CR** son las matrices anteriormente descritas que contienen el ranking, **XM** es una matriz cuyas filas se corresponden con cada una de las soluciones estables con los emparejamientos que se hacen a los motoristas e **YM** es el homólogo a **XM** pero con los cámaras**.**

Está función realizará la inicialización del proceso recursivo de prueba y error y lanzará la búsqueda de la pareja para el primer motorista. Según lo que se ha comentado, en la inicialización del proceso debe:

1º Inicializar los vectores **X** e **Y**

2º Crear las matrices **RMC** y **RCM.**

3º Lanzar la función **Prueba** para el motorista 1.

Por otra parte hay que, lógicamente, programar la función Prueba, cuyo encabezado será:

function Prueba(M,X,Y)

Siendo **M** el motorista al que se le busca pareja, **X** son los cámaras asignados a los motoristas e **Y** son los motoristas asignados a las cámaras.

Por último será necesario programar una tercera función que devuelva si un emparejamiento es o no estable. El encabezado será:

function E=Estable(M,C,X,Y)

Donde **X, Y** y **M** coinciden con los valores de entrada de la función **Prueba. C** es el cámara que se prueba a emparejar con el motorista **M** y **E** es 1 si el emparejamiento es estable y 0 si no lo es.

El resto de datos que necesitan las funciones (**MR, CR, RMC** o **RCM**), como no van a variar una vez definidos, deben guardarse como **variables globales**, tal y como se hace con los vectores que contienen los movimientos en el "**viaje del caballo**". Hay que observar que en el caso de la función de inicialización -EmparejamientoEstable(MR,CR)- no pueden definirse como variables globales **MR** y **CR**, ya que son los propios valores de entrada a la función, así que deberá utilizar otro nombre alternativo para estas dos matrices.

También se deben definir como variables globales las dos matrices de retorno **XM** e **YM,** para añadir una nueva fila cada vez que se encuentre alguna solución.